Taktgenaue Bus-Simulation mit der Transaction-Level-Modellierung

Von der Carl-Friedrich-Gauß-Fakultät

der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades

Doktor-Ingenieur (Dr.-Ing.)

genehmigte

Dissertation

von Dipl.-Ing. Robert Günzel

geboren am 16. März 1979

in Lauchhammer

Eingereicht am:	17. März 2011
Mündliche Prüfung am:	23. Juni 2011
Referent:	Prof. Dr. Ulrich Golze
Korreferent:	Prof. Dr. Rolf Drechsler

(2011)

Kurzfassung

Fokus dieser Arbeit ist die abstrakte Transaction-Level-Modellierung (TLM) von Kommunikationsstrukturen mit memory-mapped Bus-Interfaces. Dort wird der Begriff der Taktgenauigkeit untersucht, variiert und schließlich formal definiert. Darauf baut ein TLM-Modellierungsstil für taktgenaue Modelle auf, der unabhängig vom Busprotokoll ist. Dieser wird als ein Standard zur taktgenauen Modellierung von Kommunikation über memory-mapped Bus-Interfaces vorgeschlagen, und die Anwendbarkeit des Vorschlags wird untersucht.

Es wird gezeigt, wie existierende memory-mapped Bus-Interfaces mit dem Standard modelliert werden können. Dabei werden auch Möglichkeiten zur Optimierung des verwendeten SystemC-Simulators hinsichtlich der taktgenauen Modellierung diskutiert. Evaluiert wird der vorgestellte Ansatz am Beispiel praxisrelevanter memory-mapped Bus-Interfaces wie ARM AMBA, IBM CoreConnect oder OCP. Die erzielbare Simulations-Performance wird untersucht durch Vergleiche von Register-Transfer- und taktgenauer TLM-Simulation beim CoreConnect-PLB von IBM.

Abstract

This thesis focusses on abstract transaction level modeling (TLM) of communication structures with memory mapped bus interfaces. The term "cycle accuracy" is examined in detail and finally defined formally. A bus protocol independent cycle accurate TLM modeling style is built upon this definition. It is proposed as a standard for cycle accurate modeling of memory-mapped bus interface communication, and its applicability is analyzed.

The thesis shows how existing memory mapped bus interfaces can be modeled using the standard. Possible performance optimization of the SystemC simulator used is discussed. The proposed approach is evaluated using state-of-the-art memory mapped bus interfaces such as ARM AMBA, IBM CoreConnect, and OCP. By comparing the register transfer level simulation and the cycle accurate TLM simulation of IBM's CoreConnect PLB, the achievable simulation performance is examined.

Danksagung

Die vorliegende Arbeit wäre ohne Unterstützung in dieser Form nicht möglich gewesen. Besonders danken möchte ich meinem Betreuer und Mentor, Herrn Professor Dr. Ulrich Golze. Er hat mir bei der Bearbeitung meines Themas den nötigen Freiraum gelassen, stand mir aber auch immer unterstützend zur Seite, wenn es notwendig wurde. Als von Natur aus eher weitläufig erklärender Mensch habe ich von ihm – neben vielen anderen Sachen – die Eleganz und Effektivität kurzer, präziser und gern auch mathematischer Erläuterungen kennen und schätzen gelernt.

Zu ebenfalls großem Dank bin ich Dr. Mark Burton, Dr. James Aldis und Herve Alexanian verpflichtet. Mit ihnen konnte ich stets – und auch über die bloße Zusammenarbeit in der OCPIP-SLD-Working-Group hinaus – meine Ideen und Konzepte diskutieren und so wertvolles Feedback erhalten. Darüber hinaus ermöglichten sie mir als Akademiker Einblick in reale industrielle Szenarios und Probleme und erlaubten mir so eine praxisrelevante Ausrichtung meiner Forschung.

Ein weiterer Dank gilt all meinen Kollegen, dabei im speziellen Christian Schröder, und den Studenten, deren Arbeiten ich im Laufe meiner Zeit als wissenschaftlicher Mitarbeiter betreuen durfte. Sie schenkten mir stets ein offenes Ohr, hinterfragten meine Ansätze kritisch und trugen so letztendlich auch zu dem vorliegenden Ergebnis bei.

Schließlich möchte ich auch noch meiner Familie danken. Derjenige, der das Glück hat, Teil einer intakten Familie zu sein, wird wissen warum und wofür.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Die Stellen, die anderen Werken wörtlich oder sinngemäß entnommen sind, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat.

Robert Günzel

Der Autor

Robert Günzel
r.guenzel@tu-bs.de

Taktgenaue Bus-Simulation mit der Transaction-Level-Modellierung

"Fast is fine, but accuracy is everything."

– Wyatt Earp, Amerikanischer Revolverheld, 1848-1929
aus [Park08]

Inhaltsverzeichnis

1. Einleitung

Die seit fast 50 Jahren nach dem Moore'schen Gesetz explodierende Komplexität von Chips und ihrem Entwurf eröffnet fast ebenso lange die Einleitungen vieler Arbeiten zum Chip- und System-Entwurf. Diese fast schon langweiligen Wiederholungen sind jedoch weniger ein Ausdruck der Phantasielosigkeit der Autoren als vielmehr des wachsenden „Design-Gaps“ zwischen dem technisch Möglichen und dem entwurfsmethodisch Beherrschbaren. Ein Blick auf den aktuellen System-on-Chip OMAP-3430 von Texas Instruments — mit „nur“ 150 Millionen[1] statt der technisch möglichen fast drei Milliarden Transistoren[2] — verdeutlicht die Herausforderung an die Entwicklungs- und Synthesemethodik: dieses Chip-System enthält zahlreiche IP-Funktionsblöcke, die über ein Network-on-Chip miteinander verbunden sind (Abbildung 1.1).

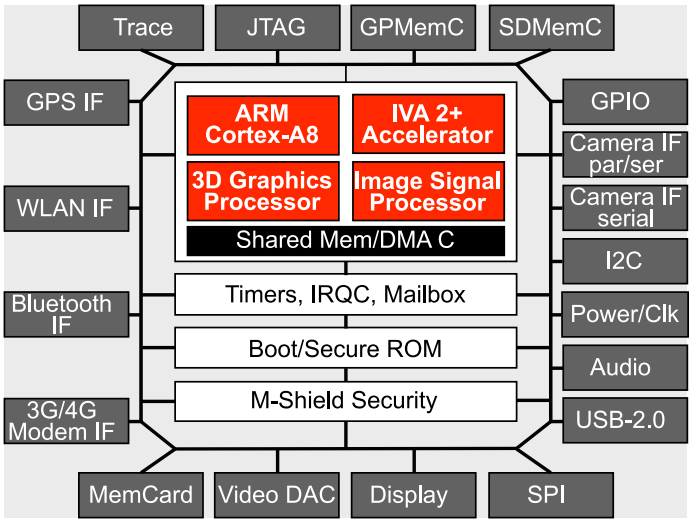


Abbildung 1.1.: OMAP-3430 Blockschaltbild (nach [Texa10]$_i$)

Dem Design-Gap versucht man heute u.a. mit einem wachsenden Maß an Abstraktion zu begegnen. In frühen Entwurfsstadien werden viele Details insbesondere auch zur chip-internen Kommunikation vernachlässigt. So entstehen virtuelle Prototypen, auf denen einerseits Software bereits lange vor der Fertigstellung der Hardware entwickelt werden kann, für die andererseits bereits auf abstraktem Niveau viele wesentliche Architekturentscheidungen effizient getroffen werden können[3].

Solche abstrakten Modelle werden oft im Rahmen der Transaction-Level-Modellierung[4] (TLM) entwickelt. Dort kann die Kommunikation mit abstrakten Transaktionen und einfa-

[1] [MWG+07]; [2] [Maxf08, Rusu09]$_i$; [3] [BuMo06, BaMP07, CCH+99, JeWo05]

chen Interfaces modelliert werden, ohne zahlreiche Ports und Signale verwalten zu müssen. Mit dem Aufkommen der TLM wuchs die Hoffnung, hochkomplexe Systeme so einfach wie Steckbausteine zusammensetzen zu können. Diese Hoffnung wurde leider nicht erfüllt.

Verschiedene Forschungsgruppen, Firmen und Konsortien brachten in guter Absicht TLM-„Standards“ auf den Weg[5]. In modernen Systems-on-Chip (SoC) stammen die einzelnen Funktionsblöcke und deren Modelle aber häufig nicht aus einer einzelnen Quelle, sondern entsprechen unterschiedlichen „Standards“. Im Beispiel des OMAP-3430 stammen der Prozessor von ARM[6], der Grafik-Prozessor von PowerVR[7] und das Network-on-Chip von Sonics[8]. Als Interfaces kommen neben AXI[9] und AHB[10] auch OCP[11] zum Einsatz. Das OMAP-3430 wird dann beispielsweise von Motorola und Nokia eingesetzt[12].

Nun könnten sich bei der OMAP-Plattform alle Beteiligten auf einen Projekt-Standard einigen, doch sind Firmen wie ARM IP-Lieferanten für verschiedene Chip-Hersteller. Daher ist es unwahrscheinlich, dass ein anderer Chip-Hersteller als Texas-Instruments den gleichen „Standard“ zu verwenden bereit ist. Die IP-Lieferanten und auch die Hersteller von TLM-Werkzeugen sahen sich damit konfrontiert, ihre IP-Modelle und Werkzeuge in vielen „Geschmacksrichtungen“ liefern zu müssen — ein logistischer und ökonomischer Unsinn. Die Wartung verschiedener Modell- und Werkzeugvarianten erfordert selbst beim Einsatz von Adaptern einen großen Aufwand.

Einen Ausweg bot nur ein TLM-Standard für alle Firmen und Projekte. So entstand der Standard TLM-2.0[13] . Fast alle an der TLM beteiligten IP-, Werkzeug- und Chip-Hersteller waren bei der Entstehung involviert, und entsprechend groß ist heute die Akzeptanz. TLM-2.0 legt TLM-Interfaces für die abstrakte Modellierung von Funktionsblöcken und Kommunikationsstrukturen mit memory-mapped Bus-Interfaces[14] fest. Die gewählte Einschränkung auf memory-mapped Bus-Interfaces ist dabei unerheblich. Bei SoCs ist nahezu jeder Funktionsblock mit einem solchen Interface ausgestattet, und die Kommunikationsstrukturen können wie beim OMAP-3430 sogar komplexe NoCs sein.

Allerdings lassen sich damit in der TLM immer noch nicht IP-Blöcke mit Bus-Schnittstellen wie Steckbausteine zusammenstecken. Als Haupthindernis gilt, dass bisher nur die zeitlich sehr abstrakte TLM standardisiert wurde. Da TLM tendenziell eine abstrakte Modellierung sein soll, drängt sich jedoch die Frage auf, ob weniger abstrakte oder gar taktgenaue TLM überhaupt notwendig ist. Dafür möchte ich die zwei wichtigsten Gründe anführen.

Erstens kann es sich bei einer abstrakten Systemmodellierung und -analyse herausstellen, dass es für eine bestimmte Hardware-Komponente keinen fertigen IP-Block gibt, so dass dieser neu entwickelt werden muss. Ausgehend vom vorhandenen abstrakten Modell wird dann der fehlende Block schrittweise solange verfeinert, bis eine klassische Hardware-

[4] [CaGa03]; [5] [SiMu01, CCG+03, GrGS05, PaDB04],[RSPF05, OCP-08, STMi05, IBM-06]$_i$; [6] [ARM-10]$_i$
[7] [PoVR10]$_i$; [8] [Soni08]$_i$; [9] [ARM-03]; [10] [ARM-99]; [11] [OCP-06]$_i$; [12] [Moto10, Klug10, Noki10]$_i$
[13] [OSCI09a]$_i$

[14] Über solche Interfaces erfolgen stets nur lesende oder schreibende Zugriffe auf Speicheradressen. Dabei kann der adressierte „Speicher“ aber auch ein IP-Block sein, der die Zugriffe als Kommandos interpretiert. Details dazu in Abschnitt 2.3.1.

Synthese möglich ist[15]. Diese Synthese-Vorgabe ist natürlich deutlich weniger abstrakt, im schlimmsten Falle entspricht sie dem Register-Transfer-Level.

Die Überführung der am wenigsten abstrakten, noch standardisierten TLM-Darstellung in eine synthetisierbare ist in einem Schritt für nicht-triviale Funktionsblöcke kaum machbar. Das Verlassen der TLM-Welt, also der Übergang von relativ einfachen Schnittstellen hin zu RTL-Schnittstellen mit hunderten von Signalen sollte aber möglichst spät im Entwurf passieren, nur leider gibt es keinen weniger abstrakten TLM-Standard. Der Entwickler muss entweder zwei wahrscheinlich völlig verschiedene TLM-Modellierungen (die standardisierte, abstrakte und eine proprietäre, taktgenaue) beherrschen, oder das Modell muss an einen Experten für die taktgenaue Modellierung übergeben werden. Der erste Fall stellt eine große Belastung des Entwicklers dar, insbesondere wenn bei der taktgenauen TLM wieder mehrere grundverschiedene proprietäre TLM-Schnittstellen existieren. Der zweite Fall ist teuer und birgt das Risiko von Missverständnissen, die bei der Übergabe der Modelle passieren.

Der zweite Hauptgrund für den Bedarf an taktgenauem TLM ist, dass die Ergebnisse einer abstrakten TLM-Simulation für diverse Fragestellungen nicht ausreichen[16]. Während im abstrakten Modell die Aussage genügen mag: „Circa 100 MHz liefern eine ausreichende Antwortzeit“, kann beispielsweise bei einem mobilen Gerät die Entscheidung zwischen 95 oder 100 MHz den entscheidenden Marktvorteil einer längeren Akkulaufzeit bedeuten. Dann benötigt man eine präzise Aussage, ob 95 MHz ausreichen.

Während solche Vorhersagen mit Register-Transfer-Modellen berechnet werden können, ist die Performance solcher RTL-Simulationen im Vergleich zur abstrakten TLM sehr gering. Ideal wäre daher ein abstraktes TLM-Modell, das trotzdem wesentliche Aussagen des präzisen RTL-Modells effizient liefert.

Genau dafür ist auch eine Standardisierung der taktgenauen TLM notwendig. Damit würden sich für die IP- und Werkzeughersteller erneut Aufwand und Kosten reduzieren. Falls ein solcher bisher fehlender Standard für taktgenaues TLM darüber hinaus auf den Mechanismen des vorhandenen abstrakten Standards TLM-2.0 aufbauen würde, falls also die Standards nahe beieinander liegen würden, müssten Entwickler nur wenig hinzulernen, um sowohl abstrakt als auch taktgenau modellieren zu können.

Ziele der Arbeit

Hauptziel meiner Arbeit ist der Vorschlag eines Standard-Interface für taktgenaue TLM von Funktionsblöcken und Kommunikationsstrukturen mit memory-mapped Bus-Interfaces in Analogie zum vorhandenen TLM-2.0.

Wie erwähnt soll ein taktgenaues TLM-Modell möglichst viel abstrahieren und immer noch gewünschte Aussagen liefern. Beides muss präzisierbar sein: Wovon wird abstrahiert? Welche Aussagen sind trotzdem möglich? Daher werde ich die taktgenaue TLM-Simulation genauer untersuchen und Möglichkeiten zur Abstraktion aufzeigen, die die Taktgenauigkeit

[15] [Donl04, Golz10],[KoHA05]$_i$; [16] [Aldi06, Aziz09, Ghen05, DBD+06, VDK+08],[Engb08]$_i$

der gewünschten Simulationsaussagen nicht gefährden. Dabei wird auch diskutiert, welche Aussagen überhaupt von Interesse sind.

Bei meinem Vorschlag für einen taktgenauen TLM-Standard werde ich nachweisen, dass die bereits in TLM-2.0 vorhandenen Mechanismen im Wesentlichen ausreichen. Die notwendigen Erweiterungen und Änderungen von TLM-2.0 werden im Detail erläutert. Neben diesen generischen Änderungen von TLM-2.0 werden auch Regeln bestimmt, wie spezielle memory-mapped Bus-Interfaces auf meine um Taktgenauigkeit erweiterte TLM-2.0 abgebildet werden können.

Schließlich wird der verwendete SystemC-Simulator für taktgenaue Modelle wesentlich optimiert und danach in praxisrelevanten Experimenten evaluiert.

Aufbau des Dokuments

Kapitel 2 führt in die Grundlagen ein. Kapitel 3 definiert die taktgenaue Simulation zunächst allgemein und präzisiert sie dann im Kontext von TLM und dort häufigen Anwendungsfällen. Kapitel 4 führt den auf TLM-2.0 basierenden Vorschlag eines taktgenauen Modellierungs-Standards ein. Anschließend werden in Kapitel 5 Optimierungen des verwendeten Simulators analysiert.

Kapitel 6 zeigt praktische Anwendungen des vorgeschlagenen Standards auf, illustriert diese anhand einer aktuellen Kommunikationsstruktur und demonstriert erzielbare Speed-Ups gegenüber klassischer, taktgenauer Modellierung auf der Register-Transfer-Ebene. Eine Zusammenfassung und ein Ausblick schließen die Arbeit ab. Die Anhänge enthalten detaillierte Beispiele und tiefergehende Erläuterungen.

Hinweise zum Schriftsatz

Fließtext ist grundsätzlich in Roman gesetzt. Ein zentraler Begriff erscheint bei erstmaliger Verwendung **fett**, und seine Bedeutung wird dabei erklärt. Genau mit dieser Bedeutung ist der Begriff für die vorliegende Arbeit reserviert, und grundsätzlich werden keine Synonyme für einen solchen Begriff verwendet. Erfolgt die Begriffserklärung relativ ausführlich und exakt oder als mathematische Definition, erhält sie eine Nummerierung (z.B. Begriffserklärung 2.23, Definition 2.42). Eine Erklärung kann aber auch „im Vorbeigehen“ relativ knapp oder implizit, gelegentlich sogar überhaupt nicht erfolgen, wenn der zu erklärende Begriff allgemein bekannt und nicht umstritten ist.

Abkürzungen werden mindestens bei ihrer ersten Nennung ausgeschrieben und anschließend in Klammern gesetzt. Handelt es sich bei der Abkürzung gleichzeitig um einen zentralen Begriff, wird beides **fett** gesetzt. Beschriftungen in Abbildungen sind serifenlos, Programmcode in `Typewriter`. Unmittelbare Bezüge auf Programmcodes oder dortige Eigennamen erscheinen ebenso in `Typewriter`. Mathematische Formeln werden *kursiv* gesetzt.

Im Beispiel [ABCD09, EFGH08a]$_i$ weist das tiefgestellte „i“ auf eine Internet-Quelle hin.

2. Grundlagen

Inhalt

2.1. Transaction-Level-Modellierung

Der Begriff der **Transaction-Level-Modellierung (TLM)** wird von Grötker in [Grot02] geprägt. Dabei handelt es sich nicht um eine klare Definition von TLM, sondern eher um die Beschreibung einer Methodik oder eines Paradigmas. Das zugrunde liegende Konzept kann aber nicht Grötker allein zugeschrieben werden. Grötker beschreibt TLM als High-Level-Modellierungsansatz, bei dem die Kommunikation zwischen Systemmodulen von deren Funktion getrennt wird. Die Funktion wird in **(TLM-)Modulen**, die Kommunikation in **(TLM-)Channels** gekapselt, auf die die Module über **(TLM-)Interfaces** zugreifen. Dadurch können Kommunikation und Funktion unabhängig voneinander und unterschiedlich abstrakt modelliert werden. Bei identischen Interfaces können sogar zwei unterschiedliche Channels gegeneinander ausgetauscht werden, ohne dass Änderungen der Module nötig sind. Dieses Konzept wurde bereits in [RoSa97] als Interface-Based-Design eingeführt, in [KMN+00] als Orthogonalization-of-Concerns verallgemeinert, und auch Gajski trennt in [GZD+00] zwischen Modulen und Channels und übernimmt in [CaGa03] dann den Begriff der TLM.

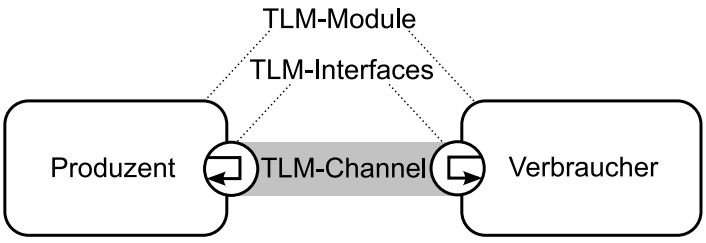


Abbildung 2.1.: Ein einfaches TLM-Modell

Abbildung 2.1 veranschaulicht die oben genannten Begriffe. Man erkennt die Module Produzent und Verbraucher, die über einen Channel verbunden sind. Die nicht notwendigerweise identischen Interfaces beider Module bestimmen, wie die Module auf den Channel zugreifen können, sie repräsentieren also die Sicht des Moduls auf den Channel.

Die Module verwenden nur die Interfaces des Channels und sind somit unabhängig vom Verhalten des Channels. Der Channel könnte ein FIFO, LIFO oder gar ein komplexes Kommunikationsprotokoll simulieren, ohne dass dies die Implementierung der Module beeinflusst.

TLM und Abstraktion

Das oben beschriebene TLM-Paradigma bezieht sich ausschließlich auf die Modellstruktur, geht aber in keiner Weise auf die angestrebte oder erzielbare Abstraktion ein. Dementsprechend stellt sich die Frage, worauf sich das „Level“ in TLM eigentlich bezieht. Dabei gehen die Meinungen weit auseinander. So definiert Pasricha in [PaDB04] die Abstraktionsebene Cycle-Count-Accurate-at-Transaction-Boundaries (CCATB)[1] als unterhalb der TLM. Dabei geht er implizit davon aus, dass die TLM bezüglich zeitlicher Abstraktion nur sehr grobe Schätzungen zu liefern im Stande ist.

Im Gegensatz dazu beschreibt Donlin in [Donl04] Anwendungen verschiedener TLM-Modelle, die in ihrer zeitlichen Abstraktion von zeitfrei bis taktgenau variieren. Dabei wird CCATB als eines dieser TLM-Modelle angesehen.

Darüber hinaus kann man einen TLM-Kanal definieren, der einen binären inneren Zustand hat und dessen Interfaces es erlauben, diesen Zustand auszulesen, zu ändern und auf Änderungen des internen Zustands zu reagieren. Ein VHDL `signal` [IEEE09] ist solch ein Kanal. Man kann und muss folglich das Register-Transfer-Level (RTL) als Teil von TLM bezeichnen.

Es wird also deutlich, dass der Terminus TLM nicht unmittelbar mit einer Abstraktionsebene gleichgesetzt werden kann. In [CaGa03] wird diese Problematik aufgegriffen, und es wird versucht, den Raum der TLM-Modelle zu klassifizieren. Dabei wird die Abstraktion eines TLM-Modells in einem zweidimensionalen Feld bezüglich der zeitlichen Abstraktion der Modulfunktion und der zeitlichen Abstraktion der Modulkommunikation festgelegt (Abbildung 2.2). Dies ist aufgrund der oben erwähnten, durch TLM bedingten Trennung von Kommunikation und Funktion möglich. In [CaGa03] wird damit für einen sog. Top-Down-TLM-Entwurf vorgeschlagen, ein vollkommen zeitfreies Modell (auch „ausführbare Spezifikation“ genannt, in Abbildung 2.2 als Punkt A bezeichnet) über die Verfeinerungen B, C, D oder B, C, E bis hin zum vollkommen taktgenauen Modell F zu überführen.

Interessanterweise wird, wenn es um Abstraktion in der TLM geht, nahezu ausschließlich die zeitliche Abstraktion diskutiert. Golze [Golz10] und auch Klingauf [Klin08] identifizieren

[1] Dabei beginnt und endet eine Transaktion zu exakt denselben Zeitpunkten wie in der taktgenauen Referenz des Modells, jedoch sind keine Aussagen über den Zustand der Transaktion zwischen diesen Zeitpunkten möglich.

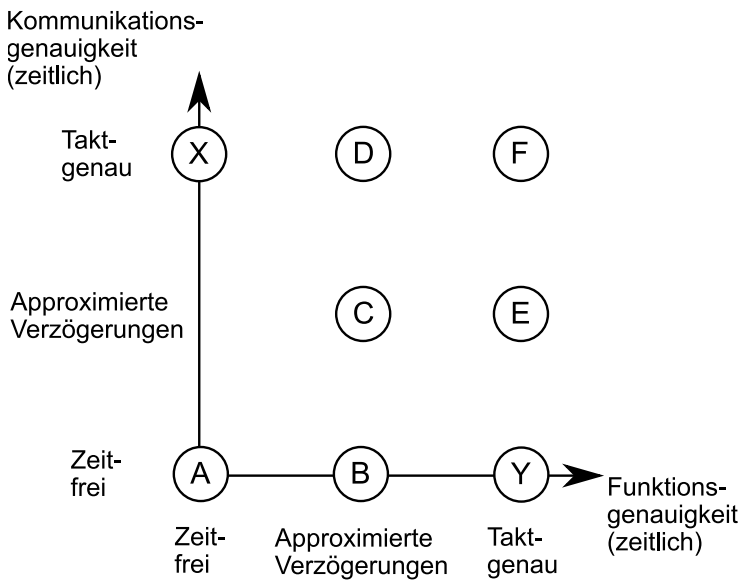


Abbildung 2.2.: Zeitliche Abstraktion bei TLM (nach [CaGa03])

neben der zeitlichen noch die Daten-, Algorithmen-, Struktur- und Kommunikationsabstraktion, während Burton in [BAGK07] die sog. Use-Case-Abstraktion nennt.

Bei der **zeitlichen Abstraktion** wird von absoluten Zeitaussagen abstrahiert. Zeit wird dann zum Beispiel in Taktschritten, Kontrollschritten oder Synchronisationspunkten gemessen. Ein Beispiel für die **Use-Case-Abstraktion** ist das Abstrahieren von Fehlerfällen. Dabei werden dann zum Beispiel Prüfsummen nicht berechnet, sodass Prüfsummenfehler nicht behandelt werden müssen. Dementsprechend wird nur Kommunikation modelliert, bei der Prüfsummenverletzungen unmöglich sind. Beispiele für die **Daten-Abstraktion** sind das Abstrahieren von Byte-Ordnungseffekten oder der Repräsentation von Daten als abstrakte Objekte. Im Rahmen der **Algorithmen-Abstraktion** wird z.B. von vorzeichenfreier Ganzzahlarithmetik in Hardware abstrahiert und die Hardwareberechnungen werden mit Hilfe von Fließkommazahlen modelliert, sodass die Algorithmen einfacher werden. Bei der **Struktur-Abstraktion** wird beispielsweise vom inneren Aufbau von Systemmodulen abstrahiert. Die **Kommunikations-Abstraktion** ist eine Spezialisierung von Zeit-, Daten- und/oder Strukturabstraktion bezogen auf Kommunikationsmodelle.

Die Autoren stimmen darin überein, dass diese verschiedenen **Abstraktionsdomänen** oft eng gekoppelt sind. So geht z.B. eine Abstraktion der verwendeten Algorithmen oft mit einer Datenabstraktion einher, während eine Abstraktion der Use-Cases oft auch eine gewisse Struktur-Abstraktion bedingt.

Die Abstraktion eines TLM-Modells definiert sich folglich als Position in einem mehrdimensionalen Raum, wobei die Dimensionen die jeweiligen Abstraktionsdomänen sind. Ein TLM-Modell ist dementsprechend gerade nicht auf einer Abstraktionsebene (level) sondern vielmehr auf einem Punkt des Abstraktionsdomänenraums angesiedelt. Ein treffenderer Begriff für TLM wäre also Transaction-Based-Modellierung, da bei Verwendung von TLM eben nicht eindeutig ein Abstraktions-Level identifiziert werden kann.

Nichtsdestotrotz hat sich der Begriff mittlerweile etabliert und wird deshalb im weiteren Verlauf dieser Arbeit verwendet.

2.2. SystemC

Die prominentesten Vertreter von Sprachen, die TLM unterstützen, sind SystemC [Ghen05, IEEE06b], SpecC [DoGG02]$_i$ [GZD+00] und SystemVerilog [IEEE05, SuDF06]. Jedoch hat sich SystemC gegen SystemVerilog und SpecC als der De-facto-Standard für TLM durchgesetzt [BaMP07], sodass sich diese Arbeit auf SystemC konzentriert.

SystemC ist eine Klassenbibliothek für C++ [ISO-03], die C++ um Konstrukte zur Systemmodellierung erweitert. Das mittels der SystemC-Bibliothek erweiterte C++ kann dann als Systemmodellierungssprache angesehen werden. Dazu gehören Konzepte für die Modellierung von Parallelität, Zeit und Systemhierarchie, wie sie auch in Hardwarebeschreibungssprachen wie Verilog [IEEE06a] oder VHDL [IEEE09] existieren. Im Gegensatz zu Verilog und VHDL kann in SystemC aber objektorientiert modelliert werden. Diese Eigenschaft hebt SystemC deutlich von den Hardwarebeschreibungssprachen ab.

SystemC war in der Version 1.0 ausschließlich für RTL-Modellierung konzipiert [OSCI00]$_i$, erst mit Version 2.0 wurden die Grundlagen für TLM geschaffen [OSCI02]$_i$. Abbildung 2.3 zeigt den Aufbau der SystemC-Bibliothek. Die dunkel unterlegten Elemente sind Bestandteil der Bibliothek seit Version 1.0, während die weiß unterlegten Elemente erst mit Version 2.0 hinzugefügt wurden.

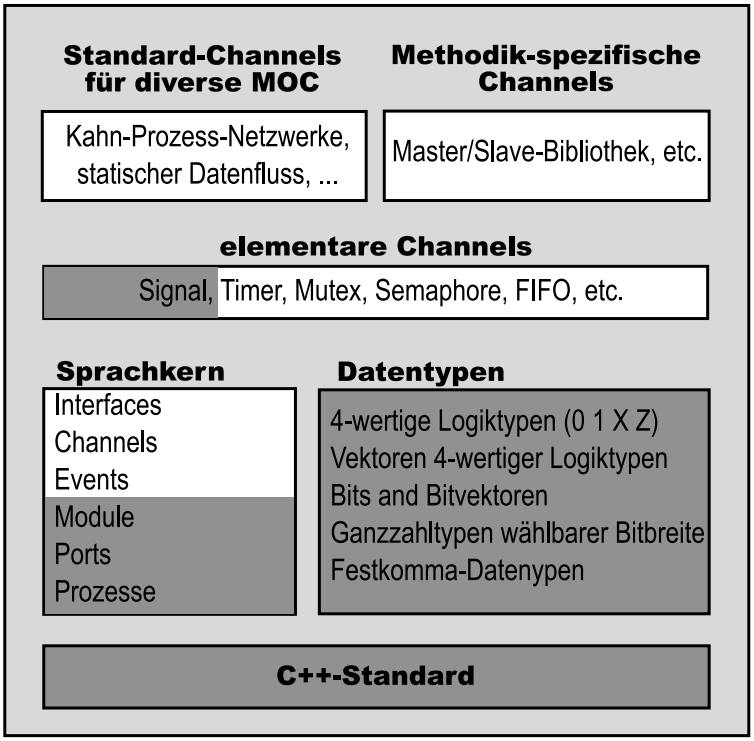


Abbildung 2.3.: Aufbau der SystemC-Bibliothek (nach [Swan01]$_i$)

Man erkennt in Abbildung 2.3 einen Schichtenaufbau. Höhere Schichten bauen ausschließlich auf den darunterliegenden auf, während die unteren Schichten auch ohne die höheren Schichten verwendet werden können. Die Datentypen sind vom eigentlichen Sprachkern von SystemC entkoppelt, sodass man die Kernelemente auch mit Standard- oder eigenen Datentypen verwenden kann.

Im Folgenden werden die Elemente des Bibliothekskerns kurz beschrieben. Ein **(SystemC-)Interface** ist eine von der Basisklasse `sc_interface` abgeleitete C++-Klasse, die lediglich pur virtuelle[2] Funktionen enthält. In einem Interface werden also die Funktionen, die später zur Kommunikation dienen, lediglich deklariert. Funktionen in einem Interface werden auch als **Interface-Method-Calls (IMCs)** bezeichnet.

Ein **(SystemC-)Channel** wird von einem Interface abgeleitet. Da das Interface pur virtuell ist, ist der Channel gezwungen, für alle im Interface deklarierten IMCs Definitionen bereitzustellen. Der Channel implementiert also mindestens ein Interface.

Ein **(SystemC-)Port** ist eine Instanz der Template-Klasse `sc_port`. Mittels des Template kann der Port für ein spezielles Interface ausgeprägt werden. Dadurch stellt der Port dann dem Besitzer des Ports die IMCs des verwendeten Interfaces zur Verfügung. Damit für das vom Port verfügbar gemachte Interface auch eine Implementierung vorhanden ist, muss ein Port vor Simulationsbeginn an einen Channel gebunden werden, der das Interface implementiert.

Ein **(SystemC-)Modul** ist eine von der Basisklasse `sc_module` abgeleitete C++-Klasse. Ein Modul kann Ports besitzen. Es verwendet zur Kommunikation dann die IMCs des vom Port angebotenen Interfaces, ohne dabei Annahmen über die exakte Implementierung zu machen. Mit anderen Worten, ein Modul, das über einen Port kommuniziert, weiß nichts vom daran angeschlossenen Channel. Hier kommt die Orthogonalisierung von Verhalten (Modul) und Kommunikation (Channel) zum tragen.

Ein **(SystemC-)Event** ist eine Instanz der Klasse `sc_event`. Ein Event kann ausgelöst werden. Das Auslösen kann augenblicklich, Deltaschritt-verzögert oder zeitverzögert werden. Die Bedeutung dieser Begriffe wird weiter unten erläutert.

(SystemC-)Prozesse werden verwendet, um Nebenläufigkeiten zu modellieren. Prozesse werden auf Events hin ausgelöst und können auf diese reagieren (dies schließt das Auslösen weiterer Events ein). SystemC unterscheidet zwischen zwei Prozesstypen: `SC_METHOD` und `SC_THREAD`. Eine `SC_METHOD` wird nach ihrem Start unterbrechungsfrei ausgeführt, kann dabei aber die Events, die ihren nächsten Start herbeiführen, neu definieren. Ein `SC_THREAD` wird nur ein einziges Mal ausgeführt, kann aber seine Ausführung unterbrechen, indem er auf Events wartet. In der Regel existiert in einem `SC_THREAD` eine Endlosschleife, in der der Prozess auf verschiedene Events wartet und reagiert.

Mit diesen Kernelementen stellt SystemC alle wesentlichen Konstrukte zur Verfügung, die zum Aufbau eines TLM-Modells notwendig sind (vgl. Abbildung 2.1).

[2] pur virtuell=virtuell ohne Implementierung

Simulation von SystemC-Modellen

Neben den in Abbildung 2.3 gezeigten Modellierungselementen definiert der SystemC-Standard auch die Semantik der Elemente bezüglich der Simulation. Die Open-SystemC-Initiative (OSCI) bietet dafür als Teil der Bibliothek auch einen Simulator an.

Wie bereits erwähnt, stellt SystemC zur Modellierung von Parallelität Prozesse zur Verfügung. Diese parallelen Prozesse verwenden das sog. kooperative Multi-Threading (erklärt z.B. in [Libe99]). Das bedeutet, dass ein Prozess aktiv die Kontrolle, also die Verwendung des Prozessors auf dem die Simulation läuft, aufgeben muss. Sie kann ihm nicht (wie beim präemptiven Multi-Threading) vom Simulator entzogen werden. Eine `SC_METHOD` erhält die Kontrolle durch ihren Start und kann diese erst beim Erreichen ihres Endes wieder abgeben. Ein `SC_THREAD` kann die Kontrolle während seiner Ausführung durch explizites Warten auf ein Event (Aufruf von `wait()`) abgeben. Er wird dann bei Eintritt des Events an exakt der Stelle fortgesetzt, an der `wait()` aufgerufen wurde.

Der SystemC-Simulator ist ein **diskreter Event-Simulator (DES)** [BCNN04], der das Verhalten eines Systems als Abfolge von Auslösungen von Events simuliert. Jedem Event ist eine **Zeitmarke** zugeordnet, und Gleichzeitigkeit wird mit Hilfe von sequentiellen Events mit gleicher Zeitmarke simuliert. Als Reaktion auf ein Event können der Zustand des Systems geändert und weitere Events ausgelöst werden. Ein Event kann nie ein Event mit kleinerer Zeitmarke auslösen. Die Abfolge der Zeitmarken der ausgelösten Events ist also immer monoton wachsend. DESs werden z.B. zur Simulation von Verilog und VHDL verwendet.

Abbildung 2.4 zeigt die Arbeitsweise des SystemC-Simulators. Nach der Initialisierung (Zustände „sc_start()“ und „Initialisierung“) sind grundsätzlich alle Prozesse ausführbar[3]. Alle ausführbaren Prozesse befinden sich in der Runnable-Queue (RQ). Nun geht der Simulator in den Zustand „Ausführbare Prozesse ausführen“ und startet zufällig einen Prozess in der RQ und entfernt diesen aus der RQ. Während der Prozess arbeitet, kann er mittels `notify()` ein Event augenblicklich auslösen, sodass alle Prozesse, die auf dieses Event warten, sofort in die RQ eingetragen werden. Er kann auch mittels `notify(SC_ZERO_TIME)` eine Deltaschritt-verzögerte Auslösung durchführen. Dies führt dazu, dass der Simulator das Event in die Liste der Deltaschritt-Auslösungen einträgt. Ähnlich verhält es sich bei `notify(sc_time)`. Hierbei wird das Event zeitverzögert ausgelöst. Der Simulator trägt das Event in die Liste der zeitverzögerten Auslösungen ein. Diese Liste ist nach den Zeitpunkten der Event-Auslösungen sortiert.

Nachdem der Prozess seine Kontrolle abgegeben hat, wird der nächste Prozess in der RQ ausgeführt. Der Simulator wiederholt dies, bis die RQ leer ist. Das Ausführen aller Prozesse der RQ wird **Deltaschritt** genannt. Nun wechselt der Simulator in den Zustand „Deltaschritt-Erhöhung“, denn der nächste Deltaschritt steht bevor. Ist die Liste der Deltaschritt-Auslösungen nicht leer, werden alle Events in der Liste ausgelöst und alle Prozesse die auf diese Events warten, werden in die RQ aufgenommen. Die Liste der Deltaschritt-

[3]Dies kann aber, wenn nötig, für einzelne Prozesse explizit unterbunden werden.

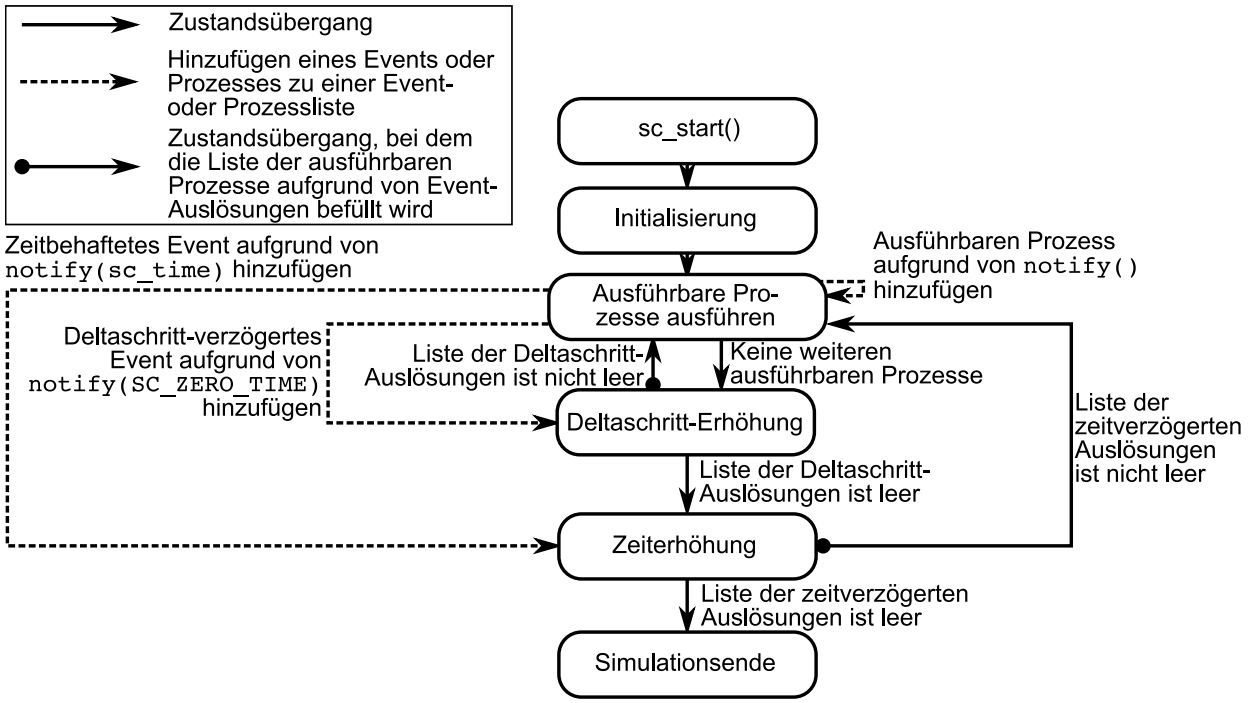


Abbildung 2.4.: Arbeitsweise des SystemC-Simulators

Auslösungen wird gelöscht. Danach wird wieder in „Ausführbare Prozesse ausführen“ gewechselt. Ist die Liste leer, wird Zustand „Zeiterhöhung“ angenommen.

Wenn die Liste der zeitverzögerten Auslösungen nicht leer ist, wird die simulierte Zeit auf den Zeitpunkt des Events gesetzt, das am wenigsten weit in der Zukunft liegt. Dann werden alle Events ausgelöst, die die Liste der zeitverzögerten Auslösungen für diesen Zeitpunkt vorgesehen hat. Alle Prozesse, die auf diese Events warten, werden in die RQ eingetragen. Alle ausgelösten Events werden aus der Liste entfernt, und der Simulator geht wieder in den Zustand „Ausführbare Prozesse ausführen“.

Es wird deutlich, dass ein simulierter Zeitschritt aus beliebig vielen Deltaschritten bestehen kann. Da der Begriff des Deltaschrittes im späteren Verlauf der Arbeit wichtig ist, soll mit dem folgenden Beispiels der Deltaschritt verdeutlicht werden.

Beispiel: Deltaschritt-Verzögerung

Gegeben sind die in Listing 2.5 dargestellten Prozesse (beide vom Typ `SC_THREAD`). Prozess `waiter` will als Reaktion auf ein Event, das von Prozess `starter` ausgelöst wird, eine Berechnung ausführen. Beide Prozesse sind zum Simulationstart ausführbar.

Wird Prozess `waiter` bei der zufälligen Auswahl des auszuführenden Prozesses vor Prozess `starter` gewählt, so wird `waiter` bis zum Aufruf von `wait(e)` fortschreiten und dadurch die Kontrolle abgeben. Er ist jetzt beim Simulator als Prozess, der auf Event `e` wartet, registriert. Kommt nun Prozess `starter` zur Ausführung, wird er Event `e` auslösen und

somit wird Prozess `waiter` vom Simulator wieder in die RQ aufgenommen, da dieser beim Simulator als auf Event `e` wartend registriert ist.

Was passiert, wenn die Prozesse in anderer Reihenfolge ausgeführt werden? Wird `starter` zuerst ausgeführt, wird er das Event `e` auslösen. Zu diesem Zeitpunkt ist aber kein Prozess als auf Event `e` wartend registriert, sodass die Event-Auslösung keinen Effekt hat. Kommt danach Prozess `waiter` zur Ausführung, wird er `wait(e)` erreichen und die Kontrolle abgeben. Er wird aber nie wieder gestartet, da Prozess `starter` bereits beendet ist.

Das Verhalten des Modells hängt also davon ab, in welcher zufälligen Reihenfolge der Simulator die Prozesse auswählt; es ist somit nicht deterministisch. Dies ist in der Regel unerwünscht.

```
1  //Ein SC_THREAD
2  void starter(){
3    do_stuff();
4    e.notify(); //Augenblickliche Ausloesung
5    do_some_more_stuff();
6  }
7
8  //Ein anderer SC_THREAD
9  void waiter(){
10   wait(e); //warte auf Event e
11   do_some_calculation();
12 }
```

Listing 2.5: Beispiel zur Prozesssynchronisation

Wie kann das Problem gelöst werden? Prozess `waiter` soll nach wie vor zum gleichen Simulationszeitpunkt ausgeführt werden wie Prozess `starter`, jedoch soll auch sichergestellt sein, dass Prozess `waiter` das Event unabhängig von der Prozessausführungsreihenfolge empfängt. Dazu kann die Deltaschritt-Verzögerung verwendet werden. Das Event muss im Prozess `starter` mittels `e.notify(SC_ZERO_TIME)` ausgelöst werden. Dadurch wird das Event noch nicht sofort ausgelöst, sondern nur zum Auslösen im Simulatorzustand „Deltaschritt-Erhöhung“ vorgemerkt. Dieser wird erst erreicht, wenn beide Prozesse ausgeführt wurden, da sie beide in der RQ waren. Ist jetzt der Simulator im Zustand „Deltaschritt-Erhöhung“, ist Prozess `waiter` auf jeden Fall als auf Event `e` wartend beim Simulator registriert, da er ja bis zu `wait(e)` ausgeführt wurde. Nun wird er also unabhängig von der Prozessreihenfolge das Event bemerken und fortgesetzt werden. Da es sich um eine Deltaschritt-Verzögerung handelte, ist auch die simulierte Zeit noch nicht erhöht worden, das Modell verhält sich also in jedem Fall wie gewünscht.

2.3. TLM-2.0

Wie schon in Abschnitt 2.1 erwähnt, gehen die Interpretationen und dementsprechend auch die Anwendungsgebiete von TLM weit auseinander. Das wäre prinzipiell nicht problematisch, solange die Interpretationen zumindest innerhalb eines Modells identisch sind. Dies ist der Fall, wenn das gesamte Modell „aus einer Hand” stammt. Aktuelle System-on-Chip

(SoC) setzen sich aber oft aus Elementen von verschiedenen Herstellern, sog. **Intellectual Property (IP)**-Blöcken, zusammen [CCH+99, Lin-06]. Teile des Systems werden eingekauft und die Modellentwicklung von (mehreren) Dritten übernommen. Ein Systemmodell kann sich daher heute aus Modellen vieler verschiedener Quellen zusammensetzen. Von zentraler Bedeutung ist in solchen Fällen, dass die Modelle unabhängig von ihrer Herkunft problemlos zu einem Gesamtmodell zusammengesetzt werden können, dass also **Interoperabilität** vorhanden ist. Herrscht bei diesen Quellen kein Einverständnis bezüglich der Interpretation von TLM und den verwendeten Abstraktionen, entsteht ein Interoperabilitätsproblem [KBGG07]$_i$.

Dieses Problem wurde sowohl in der Forschung als auch in der Industrie identifiziert. Der erste Versuch eine TLM-Standardisierung für SystemC seitens der Industrie war der TLM-1.0-Standard [RSPF05]$_i$. Dieser konnte sich aber nicht durchsetzen, da er keinerlei Datenformat oder -abstraktion definierte. Alle Standard-Klassen waren Template-Klassen, in denen der Template-Parameter die Datenstruktur des Interfaces war. Aus diesem Grund konnten zwei Module zwar TLM-1.0-konform sein, aber trotzdem untereinander inkompatibel bleiben [Edwa07].

Seitens der Forschung wurde mit GreenBus [Klin08, KGB+06] erstmals ein Standardvorschlag geschaffen, dessen Interfaces auch datenseitig fixiert waren, jedoch immer noch die Möglichkeit für diverse Erweiterungen bot. Die Vorteile einer solchen Interoperabilität wurden auch von der OSCI erkannt, und viele Erkenntnisse, die mit GreenBus gewonnen wurden, flossen in den neuen Industriestandard TLM-2.0 [OSCI09a]$_i$ ein.

Der TLM-2.0-Standard findet seitens der Industrie hohen Zuspruch [OSCI09b]$_i$, da er erstmals Interoperabilität über Firmengrenzen hinweg ermöglicht. Da sich diese Arbeit der Anwendung des TLM-2.0-Standards für taktgenaue Simulationen widmet, wird im Folgenden der Standard etwas genauer dargestellt. Vorher ist aber noch die Klärung des Terminus memory-mapped Bus-Interface, eines zentralen Begriffes im TLM-2.0-Standard, wichtig.

2.3.1. Memory-mapped Bus-Interface

Der Terminus memory-mapped Bus ist in verschiedenen Patenten (z.B. [Bres06, MEDB95]$_i$) und Standards (z.B. [OSCI09a]$_i$) zu finden, es existiert aber keine zitierbare Definition. Er baut auf mehreren anderen Begriffen auf, die im folgenden erläutert werden.

Memory-Mapped-Input-Output (MMIO) ist ein Ein-/Ausgabeschema, bei dem Teile des System-Adressraumes anstelle von Speicher an Ein-/Ausgabe-Einheiten zugewiesen werden. Schreib- und Lesezugriffe auf diese Adressen werden als Kommandos für die Ein-/Ausgabe-Einheiten interpretiert (vgl. [PaHe08]).

Ein Processor-Memory-Bus (PMB) verbindet einen Prozessor oder ähnliches mit einem Speicher. Der Shared-Processor-Memory-Bus (SPMB) ist ein PMB, an dem mehrere Prozessoren oder ähnliches angeschlossen sind. Ein Arbiterierungsmechanismus stellt sicher, dass stets nur ein Prozessor auf gemeinsam genutzte Teile der Busstruktur zugreifen kann (vgl. [JeWo05]).

Begriffserklärung 2.6 :
Ein **memory-mapped Bus (MMB)** ist ein SPMB, an dem mittels MMIO neben Systemspeicher auch diverse Ein-/Ausgabegeräte angeschlossen sein können. Module, die aktiv Lese- oder Schreibzugriffe starten, werden als **Master** bezeichnet, Module, die auf solche Lese- und Schreibzugriffe reagieren als **Slave**. Die Rolle einen Masters ist nicht ausschließlich Prozessoren vorbehalten. Ein Modul kann gleichzeitig Master und Slave sein.

Eine **memory-mapped Bus-Interface (MMBIF)** ist ein Interface zwischen einem MMB und einem angeschlossenen Modul.

Abbildung 2.7 illustriert die Unterschiede zwischen PMB, SPMB und MMB. In Abbildung 2.7c sind darüber hinaus noch die Rollen am Bus mit angegeben. Man beachte, dass der Direct-Memory-Access (DMA)-Controller sowohl Master als auch Slave ist, da er von Prozessoren als Slave DMA-Kommandos erhält und anschließend als Master die DMA-Transfers durchführt.

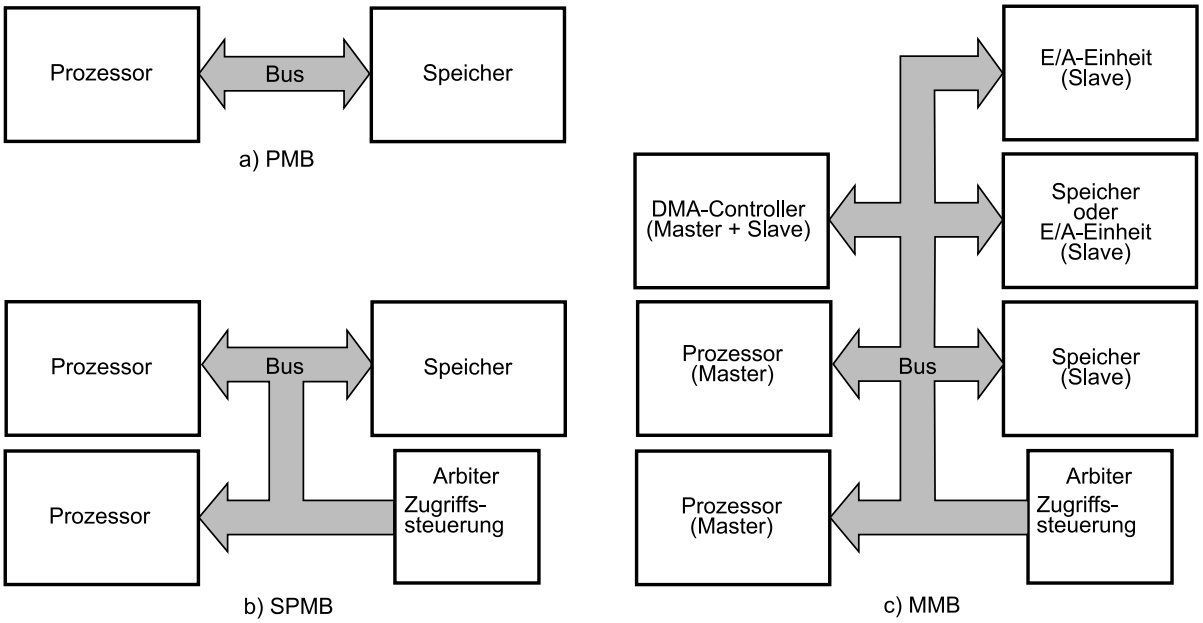


Abbildung 2.7.: Beispiele für PMB, SPMB und MMB

Die populärsten Vertreter für MMBIF sind die Advanced-Microcontroller-Bus-Architecture (AMBA) in Version 2.0 [ARM-99], das Advanced-eXtensible-Interface (AXI) [ARM-03], Wishbone [Herv02]$_i$, CoreConnect [IBM-99], das Open-Core-Protocol (OCP) [OCP-06]$_i$ und der STBus [STM-07]. Eine Vielzahl von IP besitzt MMBIF, und man kann davon sprechen, dass MMBIF den Normalfall in eingebetteten Systemen darstellen. Es ist wichtig, dass Begriffserklärung 2.6 die physikalische Umsetzung des MMB weitestgehend frei lässt. Unter Verwendung eines zentralen Arbiters wie in Abbildung 2.7c entsteht ein „klassischer“ Bus, während bei Wahl eines dezentralen Arbitrierungsmechanismus auch ein Network-on-Chip (NoC) entstehen kann. Existiert nur ein Master und ein Slave an einem MMB, kann dieser zu einer extrem simplen Struktur wie einem FIFO degenerieren.

Bei all diesen MMBIF kann die Kommunikation in **Busphasen**[4] eingeteilt werden. Eine solche Phase besteht dabei aus einer Gruppe von Signalen und den Festlegungen, wann und wie eine Phase beginnt und endet. In Anhang B werden die verschiedenen Phasentypen aufgelistet und in Anhang D, Abbildung D.1, sind als Beispiel die Phasen dargestellt, die bei einer Kommunikation über den On-Chip-Peripheral-Bus (OPB) am Interface eines OPB-Masters durchlaufen werden. Grüne Ellipsen markieren einen Phasenstart, rote eine Phasenende. Die Ellipsen umschließen stets alle Signale, die zur jeweiligen Phase gehören.

2.3.2. Inhalt des TLM-2.0-Standards

Der Fokus von TLM-2.0 liegt auf größtmöglicher Interoperabilität von Modellen, die Daten über MMBIF austauschen. Abbildung 2.8 zeigt exemplarisch ein TLM-2.0-Modell, in welchem verschiedene Module miteinander kommunizieren. Mit TLM-2.0 sollen die einzelnen Module des Systems unabhängig voneinander von verschiedenen Entwicklern erstellt werden können und anschliessend direkt interoperabel sein bzw. mit sehr einfachen Adaptern verbunden werden können.

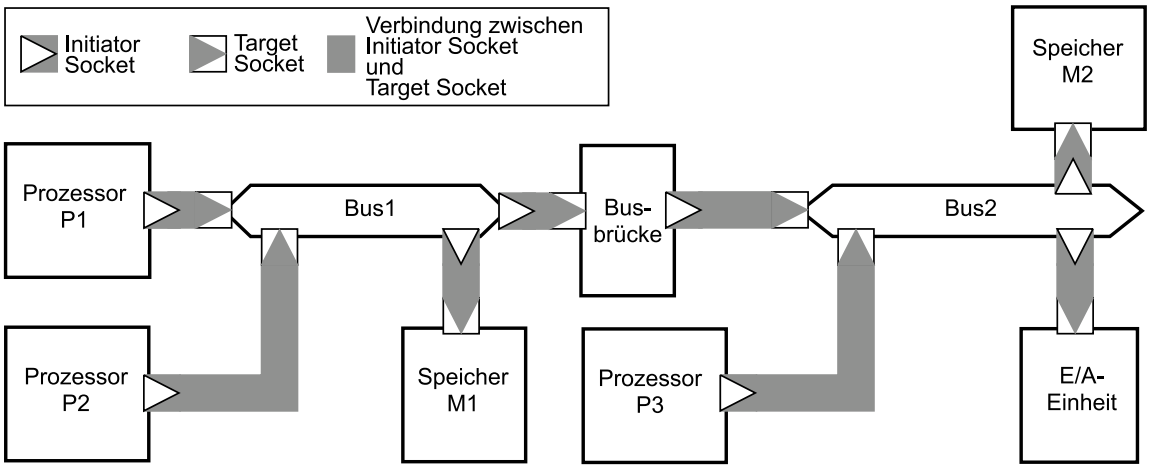


Abbildung 2.8.: Ein Beispiel eines TLM-2.0-Modells

Dementsprechend ist TLM-2.0 ein zweiteiliger Standard: Er besteht zum einen aus einer Sammlung von Interfaces, die genau wie TLM-1.0 Template-Klassen sind, sodass die eigentlich zu transportierenden Daten nicht festgelegt sind. Im zweiten Teil definiert der Standard Datentypen und Kommunikationsabläufe, die zur Modellierung von MMBIF verwendet werden können. Die Fokussierung auf MMBIF resultiert aus der Tatsache, dass aktuelle SoC zur Kommunikation fast ausschließlich MMBIF verwenden. Wie bereits erwähnt, sind dies meist CoreConnect, AMBA, STBus, Wishbone oder OCP, wobei speziell die AMBA- und OCP-Interfaces auch für NoC-Architekturen im Einsatz sind [Arte09, Soni08]$_i$. Dementsprechend groß ist auch die Anzahl existierender IP mit MMBIF, für die TLM-2.0 angewendet werden kann.

[4]Siehe auch Abschnitt 3.2.1.

Zentrale Begriffe und beide Teile des Standards werden im Folgenden kurz erläutert.

Das **Payload** ist ein Datencontainer, der alle Daten (nicht notwendiger Weise gleichzeitig) enthält, die im Rahmen der Kommunikationen ausgetauscht werden. In [Klin08] existiert ein konzeptionell identisches Konstrukt als Transaction-Container.

Eine **Transaktion** ist der dynamische Vorgang, der einen kompletten Datenaustausch repräsentiert. Jeder Transaktion ist genau ein Payload zugeordnet, welches aber während einer Transaktion mehrfach übermittelt und verändert werden kann.

Ein **Initiator** ist ein Systemmodul, das die TLM-2.0-Kommunikation initiiert. Dieses Modul hält das eigentliche Payload und übermittelt lediglich eine Referenz oder einen Zeiger auf das Payload. Es wird in TLM-2.0 nicht von einem Master gesprochen, da nicht jeder reale Bus-Master als Initiator modelliert wird (siehe unten).

Ein **Target** ist ein Systemmodul, das das abschließende Ziel des Payloads repräsentiert. Ein Target wird ein Payload nicht weiterleiten. Es kann lediglich dem Modul, von dem es das Payload empfangen hat, antworten. Es wird in TLM-2.0 nicht von einem Slave gesprochen, da nicht jeder reale Bus-Slave als Target modelliert wird (siehe unten).

Ein **Interconnect** ist ein Systemmodul, das Payloads empfängt und weiterverteilt. Es ist weder Initiator noch Target[5]. Es hat sehr eingeschränkte Zugriffsrechte auf das Payload. In einem realen System kann ein Interconnect sowohl Bus-Master als auch Bus-Slave (z.B. eine Busbrücke) sein, ist aber im Modell weder Initiator noch Target. Zur Vermeidung von Verwechslungen werden in TLM-2.0 daher die Begriffe Master und Slave nicht verwendet.

Ein **Transaktionspfad** (oder auch nur „Pfad“) ist die Folge von Modulen, die ein Payload bis zum Erreichen des Targets passiert hat. Ein Pfad besteht immer aus genau einem Initiator, einer beliebigen Anzahl Interconnects und genau einem Target.

Ein **Socket** ist ein C++-Objekt, das ein MMBIF repräsentiert. Es kann als Abstraktion von den Signalen des Bus-Interfaces eines Moduls angesehen werden. Ein Master-MMBIF wird durch einen Initiator-Socket, ein Slave-MMBIF durch einen Target-Socket ersetzt. Es ist wichtig, dass ein Interconnect sowohl einen Initiator- als auch einen Target-Socket besitzt, obwohl es weder Initiator noch Target ist.

Abbildung 2.8 veranschaulicht einige der oben gelisteten Begriffe: Es existieren drei Initiatoren (die Prozessoren P1, P2 und P3), drei Targets (M1, M2 und Ein/Ausgabe-Einheit) und drei Interconnects (Bus1, Bus2 und Busbrücke). Die Busbrücke ist im realen System zwar ein Bus-Master für Bus2, im TLM-2.0-Modell aber ein Interconnect, da sie Transaktionen von Bus1 direkt an Bus2 weiter leitet. Ein Beispiel für einen möglichen Transaktionspfad entsteht, wenn Prozessor P2 mit Speicher M2 kommuniziert. Der Pfad ist dann P2→Bus1→Busbrücke→Bus2 →M2.

TLM-2.0-Interfaces

TLM-2.0 definiert vier verschiedene Interfaces, die sich in ihrer Anwendung stark unterscheiden. Diese sind das Blocking-Transport-Interface (BTI), das Direct-Memory-Interface

[5]Ein simpler Payload-Fifo ist folglich auch ein Interconnect.

(DMI), das Debug-Transport-Interface (DTI) und das **Non-blocking-Transport-Interface (NBTI)**. BTI und DMI sind für eine sehr abstrakte Modellierung definiert worden, die im TLM-2.0-Standard Loosely Timed (LT) genannt wird, in [Klin08] als Programmer's View (PV) und in [KoHA05]$_i$ als TL4 bezeichnet wird.

Das DTI ist dafür gedacht, Transaktionen auch bei angehaltener Simulation durchführen zu können. Da die Simulation pausiert, kann über das DTI weder Zeitverhalten modelliert werden, noch können Prozesswechsel erfolgen. Es kann also nicht zur Modellierung von Systemverhalten verwendet werden. Es soll aussschließlich von Debug-Werkzeugen verwendet werden, die bei angehaltener Simulation immer noch Daten der simulierten Module mittels Transaktionen auslesen oder ändern wollen.

Das NBTI wurde für weniger abstrakte Modellierung entwickelt, die der TLM-2.0-Standard als Approximately Timed (AT), [Klin08] als Bus-Accurate (BA) und [KoHA05]$_i$ als TL3 bezeichnet. Da sich diese Arbeit mit der Verwendung des TLM-2.0-Standards für taktgenaue Modellierung befasst, ist dieses Interface von zentraler Bedeutung und wird darum etwas genauer erläutert; es enthält die beiden folgenden Funktionen:

```
virtual tlm_sync_enum nb_transport_fw(TRANS& trans, PHASE& phase,
                                      sc_core::sc_time& time)= 0;
virtual tlm_sync_enum nb_transport_bw(TRANS& trans, PHASE& phase,
                                      sc_core::sc_time& time)= 0;
```

Beide Funktionen haben eine identische Signatur mit identischer Semantik. Der Unterschied liegt darin, dass `nb_transport_fw` stets von einem Initiator-Socket zu einem Target-Socket aufgerufen wird, während `nb_transport_bw` von einem Target-Socket zu einem Initiator-Socket geht. Im Folgenden wird nur noch von `nb_transport` gesprochen, wenn für beide Funktionen das Gleiche gilt, bei Unterschieden wird explizit das `fw` bzw. `bw` erwähnt.

`nb_transport` ist, wie sein Name andeutet, als nicht-blockend definiert, das heißt in der Implementierung von `nb_transport` darf unter keinen Umständen `wait` aufgerufen werden. Die Bedeutung der Argumente von `nb_transport` ist wie folgt:

`trans` : Der Typ des Arguments wird über den Template-Parameter `TRANS` des NBTI festgelegt. Es handelt sich bei dem Argument um das Payload. Es wird als Referenz, nicht als Kopie übergeben.

`phase` : Der Typ des Arguments wird über den Template-Parameter `PHASE` des NBTI festgelegt. Das Argument bestimmt die aktuelle Position im zeitlichen Ablauf der Kommunikation. Es wird Phase genannt, da es vorrangig dafür verwendet wird, um Start- und Endzeitpunkte von Busphasen (siehe Abschnitt 2.3.1) zu markieren.

`time` : Das Argument bezeichnet den zeitlichen Abstand von der aktuellen simulierten Zeit, zu dem die als `phase` angegebene Phase als gültig angesehen werden kann. Auch dieses Argument wird als Referenz übergeben.

Der Rückgabewert von `nb_transport` ist ein Aufzählungstyp[6] und kann nur drei verschiedene Werte annehmen:

`TLM_ACCEPTED` : Der Empfänger von `nb_transport` hat (bisher) in keiner Weise auf den Aufruf reagiert; die Werte von `trans`, `phase` und `time` sind unverändert.

`TLM_UPDATED` : Der Empfänger von `nb_transport` hat auf die übermittelte Phase reagiert. `trans`, `phase` und auch `time` können verändert sein. Eine Veränderung von `time` bezeichnet dabei den zeitlichen Abstand von der aktuellen simulierten Zeit, zu dem diese Reaktion als gültig angesehen werden kann.

`TLM_COMPLETED` : Der Empfänger von `nb_transport` hat auf die übermittelte Phase reagiert und betrachtet aus seiner Sicht die gesamte Transaktion als abgeschlossen. Die Werte von `trans` und `time` können verändert sein; `phase` hat keine Bedeutung mehr. Eine Veränderung von `time` ist dabei genau wie bei `TLM_UPDATED` zu deuten.

Die Template-Parameter `TRANS` und `PHASE` werden an die Interfaces über eine sog. **Types-Class (TC)** übergeben[7]. Eine solche TC hat die in Listing 2.9 gezeigte Form. Der Template-Parameter `TRANS` wird dabei stets auf `tlm_payload_type`, der Parameter `PHASE` auf `tlm_phase_type` gesetzt. Im Beispiel in Listing 2.9 würde als Payload also `my_payload_type` und als Phase `my_phase_type` verwendet werden.

```
1 struct my_tlm2_types_class
2 {
3   typedef my_payload_type tlm_payload_type;
4   typedef my_phase_type   tlm_phase_type;
5 };
```

Listing 2.9: Ein Beispiel für eine TLM-2.0-Types-Class

TLM-2.0-Base-Protocol

Da es sich bei den TLM-2.0-Interfaces um generische Template-Klassen handelt, definiert der TLM-2.0-Standard ein spezielles Protokoll für MMB, das **Base-Protocol (BP)**. Dieses legt die Datentypen für die Template-Parameter fest, sodass ein vollständig definiertes Interface für MMB (siehe Abschnitt 2.3.1) entsteht. Wie bereits in Abschnitt 2.3.2 erwähnt, muss zum Festlegen der Datentypen eine Types-Class (TC) definiert werden. Listing 2.10 zeigt die TC des BP.

```
1 namespace tlm{
2 struct base_protocol_types
3 {
4   typedef tlm_generic_payload tlm_payload_type;
5   typedef tlm_phase   tlm_phase_type;
6 };
7 }
```

Listing 2.10: TLM-2.0-Types-Class für MMB

[6] Eine Ganzzahl mit beschränktem Wertebereich (`enum`).
[7] Details zum entsprechenden Mechanismus können [OSCI09a]$_i$ entnommen werden.

Das bedeutet also, dass für MMB Payloads vom Typ `tlm::tlm_generic_payload` und Phasen vom Typs `tlm::tlm_phase` verwendet werden. Dieser Payload-Typ wird im Folgenden als **Generic Payload (GP)**, der Phasen-Typ als **TLM-Phase** bezeichnet.

Das GP ist eine Klasse, die verschiedene Attribute zur Modellierung von MMB wie Adresse, Kommando (Lesen oder Schreiben), Daten und Byte-Enables enthält. Die TLM-Phase ist ein erweiterbarer Aufzählungstyp, dessen vordefinierter Wertebereich mit `BEGIN_REQ`, `END_REQ`, `BEGIN_RESP` und `END_RESP` gegeben ist.

GP und TLM-Phase allein ermöglichen noch keine Kommunikation bzw. Interoperabilität. Dazu bedarft es noch der Festlegung der möglichen Phasenabläufe und der Definition, wann und wie die verschiedenen Attribute des GP verändert werden dürfen. Auch dies legt der TLM-2.0-Standard im Rahmen des BP fest. Abbildung 2.11 auf der nächsten Seite zeigt alle erlaubten Abläufe im BP. Man erkennt, dass es sich um ein Protokoll mit nur zwei Busphasen (Request und Response) handelt, wobei der Beginn der Response `BEGIN_RESP` das Ende des Requests impliziert. Innerhalb einer Transaktion können die Phasen also nicht überlappen.

Abbildung 2.11 stellt einen zyklenfreien Graphen dar. Demnach kann jede Phase (Request und Response) nur einmal pro Transaktion stattfinden, unabhängig von der Menge der übertragenen Daten. Das BP kann also der BA-Abstraktionsebene (vgl. [Klin08]) beziehungsweise der TL3-Abstraktionsebene (vgl. [KoHA05]$_i$) zugeordnet werden. Im TLM-2.0-Standard wird das BP zusammen mit `nb_transport` als Approximately Timed (AT) bezeichnet.

Da das BP als Basis für andere Protokolle gedacht ist, stellt das GP einen Erweiterungsmechanismus bereit. Damit ist es möglich, dem GP Zeiger auf Objekte hinzuzufügen, wenn diese Objekte von der Klasse `tlm_extension` abgeleitet sind. Darüber hinaus kann auch der Wertebereich der TLM-Phase erweitert werden, um mehr Busphasen hinzuzufügen. Werden diese Erweiterungen derart genutzt, dass die Erweiterungen zwingend für die korrekte Funktion eines Moduls notwendig sind, muss eine neue TC definiert werden. Dadurch kann dann das Modul nicht mehr direkt an ein nicht-erweitertes BP Modul gebunden werden, und ein Adapter wird notwendig. Dieses Vorgehen stellt sicher, dass Module, die die `tlm_base_protocol_types` TC verwenden, interoperabel sind.

Der Vorteil, GP und TLM-Phase mit einer eigenen TC zu verwenden, liegt darin, dass ein Adapter dann lediglich die Erweiterungen behandeln muss, während der Rest des GP unverändert bleibt und keine Kopien des GP notwendig werden. Dadurch können effiziente Protokolladapter entwickelt werden. Ein Beispiel in Anhang A verdeutlicht dieses Konzept.

2.4. Zusammenfassung

TLM ist ein unscharfes Paradigma, für das keine präzise Definition existiert und das einen großen Spielraum bezüglich der exakten Verwirklichung bietet. Die Modellierungssprache SystemC bietet alle notwendigen Mechanismen, um das TLM-Paradigma umzusetzen, lässt aber nach wie vor offen, wie von welchen Systemeigenschaften abstrahiert werden kann.

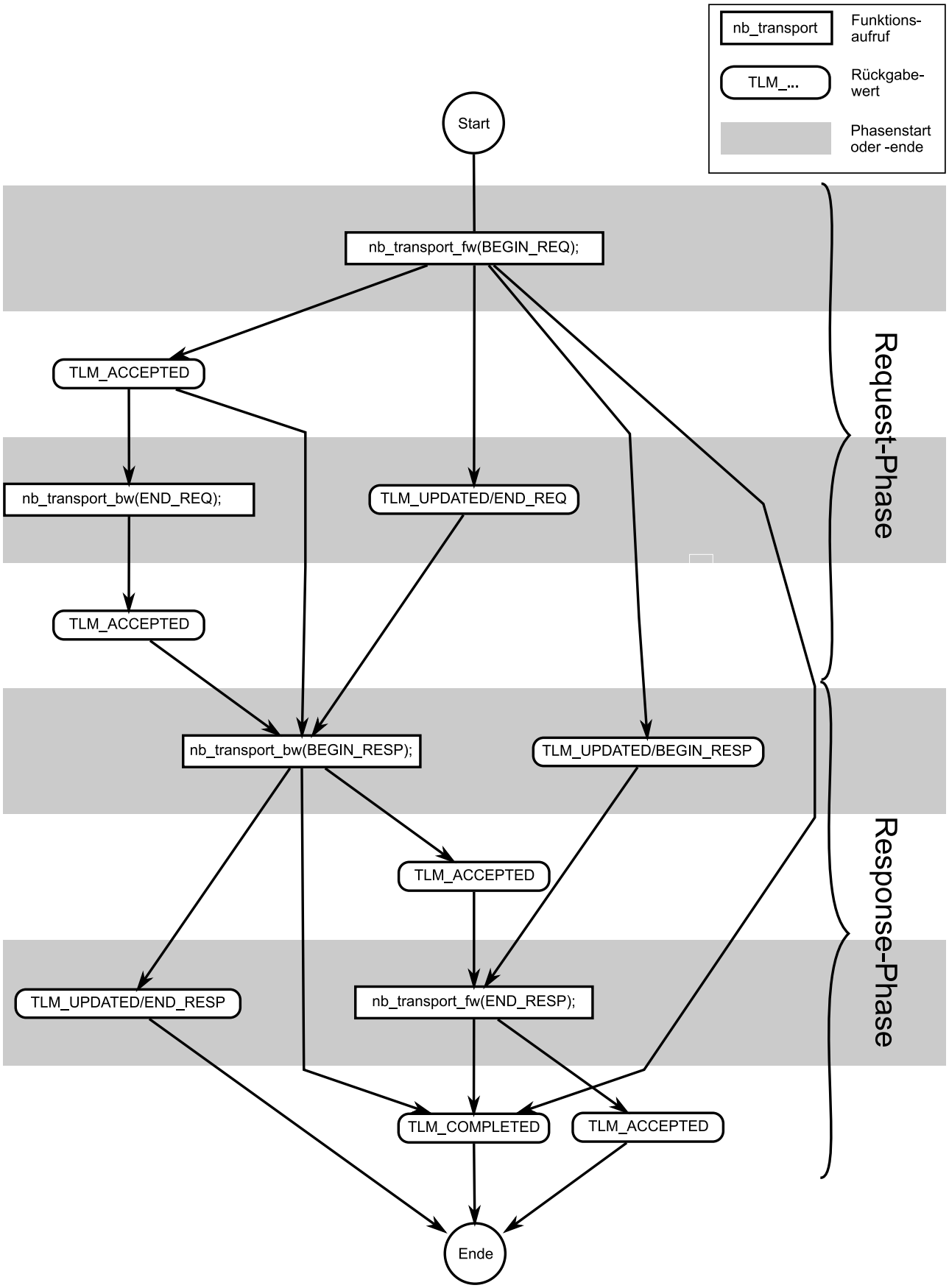


Abbildung 2.11.: Legale Abläufe im Base-Protocol (nach [OSCI09a]$_i$)

Diese große Spannweite von TLM im Abstraktionsraum stellt einerseits einen großen Vorteil dar, da man ein TLM-Modell und seine Abstraktion für die gewünschte Anwendung anpassen kann. Dies ist andererseits aber auch ein Nachteil, da sich viele verschiedene Methodiken und Stile in der Industrie herausgebildet haben und so einen firmenübergreifenden IP-Austausch erschweren. Unterschiedliche Auffassungen von Abstraktion führen darüber hinaus zu Simulationsergebnissen, die in ihrer Aussagekraft nur schwer zu beurteilen sind.

Mit dem TLM-2.0-Standard hat sich erstmals ein industrieweit anerkannter TLM-Standard für MMB etabliert. Dieser Standard definiert zwei Kommunikations-Abstraktionen bzw. „Coding Styles“: Loosely Timed (LT) und Approximately Timed (AT), ersterer für sehr abstrakte Modelle, vergleichbar mit der PV-Abstraktion ([Klin08]), letzterer für Modelle mit geschätzten Timings, vergleichbar mit der BA-Abstraktion (wiederum [Klin08]).

Während also im abstrakten Modellierungsbereich Fortschritte hinsichtlich der standardisierten Terminologie und Anwendung von TLM gemacht wurden, bleiben hinsichtlich der weniger abstrakten Anwendung von TLM noch Fragen offen. Wo im Abstraktionsraum kann ein taktgenaues TLM angesiedelt werden? Kann der TLM-2.0-Standard für solche taktgenaue Kommunkations-Modellierung eingesetzt werden? Welche Vorteile des TLM im Allgemeinen und von TLM-2.0 im Speziellen können auf der taktgenauen Ebene noch zum Tragen kommen? Und welchen Einfluss hat die diskrete Event-Simulation mit SystemC auf die taktgenaue TLM-Modellierung?

In [MCRB09] wird anhand eines einfachen Protokolls gezeigt, dass sich TLM-2.0 zur „Bus-Cycle-Accurate“-Simulation eignet. Dabei wird dieser Begriff nicht präzisiert, und der AT-Modellierungsstil wird unverändert und unreflektiert eingesetzt. Keine der oben genannten Fragen wird in [MCRB09] beantwortet, während meine Arbeit versucht, diesen auf den Grund zu gehen.

3. Taktgenaue Transaction-Level-Modellierung

Inhalt

3.1. Taktgenaue Simulation

Wie bereits in Kapitel 2 erläutert, kann TLM keiner Abstraktionsebene zugeordnet werden. Ein Modell kann hinsichtlich verschiedener Domänen unterschiedlich abstrakt sein. Ein TLM-Modell befindet sich nicht auf einer Abstraktionsebene, sondern auf einer Abstraktionskoordinate. Folglich muss die Frage beantwortet werden, wo die „taktgenaue“ Koordinate im Abstraktionsraum zu finden ist. Taktgenau bezieht sich offenbar auf die zeitliche Abstraktion. Aufgrund der Orthogonalisierung von Funktion und Kommunikation in TLM kann dies bedeuten, dass die Kommunikation, die Funktion oder beide taktgenau sind. Bezogen auf das in [CaGa03] für die zeitliche Abstraktionsdomäne vorgeschlagene Koordinatensystem (Abbildung 2.2 auf Seite 7) sind dies die Punkte X, D, F, E oder Y. Abhängig von der gewünschten zeitlichen Genauigkeit von Funktion und/oder Kommunikation können zusätzlich noch Daten-, Struktur-, Use-Case- und Algorithmenabstraktionen vorgenommen werden.

Im Rahmen dieser Arbeit wird untersucht, wie ein Standard für taktgenaue TLM-Kommunikationsmodellierung aussehen kann. Solch ein Standard kann weder die zeitliche Genauigkeit der Funktion noch die der Kommunikation erzwingen. Das hängt von der Art der Verwendung des Standards ab, er muss aber in der Lage sein, eine taktgenaue Simulation zu ermöglichen. Um die Erfüllung dieser Anforderung belegen zu können, muss definiert werden, was eine taktgenaue Simulation ist.

Zuerst muss die Referenz, an der Taktgenauigkeit gemessen werden soll, festgelegt werden. Diese Arbeit fokussiert sich auf Kommunikationsstrukturen mit MMBIF, da diese im embedded Bereich aktuell den Standard darstellen (siehe auch Abschnitt 2.3.1)[1]. Dies sind

[1]Dies ist auch ein Grund, warum sich TLM-2.0 auf MMBIF fokussiert (siehe Abschnitt 2.3).

meist klassische Busse, können aber auch simple Strukturen wie FIFOs oder komplexe Strukturen wie NoCs sein (vgl. Abschnitt 2.3.1). Im Folgenden nenne ich solche Strukturen nur Kommunikationsstrukturen.

Definition 3.1 :
Eine **Kommunikationsstruktur** $KS = (KA, Per)$ ist durch folgende Bedingungen und Bezeichnungen spezifiziert:

$KA = (Z, z_0, I, O, \delta, \lambda)$ ist ein **Kommunikationsautomat** genannter Mealy-Automat. Die Mengen Z, I und O enthalten die **Zustände**, **Inputs** bzw. **Outputs**. $z_0 \in Z$ ist der **Anfangszustand**.
$\delta : Z \times I \to Z$ und $\lambda : Z \times I \to O$ sind die **Übergangs-** bzw. **Ausgabefunktion**, die für jeden **Input-Ablauf** $\overline{i} = (i_1, ...) \in I^*$ einen **zugehörigen Output-Ablauf** $\overline{\lambda}(\overline{i}) \in O^*$ festlegen.

$Per = (M, S, PN, P, pz)$ ist die **Peripherie** mit den Mengen M von Mastern, S von Slaves, PN von **Portnamen**, $P \subset PN \times \mathbb{N} \times \{e, a\}$ von **Ports** sowie einer **Portgruppen-Zuordnung** $pz : M \cup S \to 2^P$, deren Wertebereich P disjunkt zerlegt. Dabei setzt sich ein Port (pn, bw, dir) zusammen aus dem Portnamen pn, der **Bitbreite** bw und der **Portrichtung** dir; dies führt zu einer Zerlegung von P in **Eingabeports** $P_e = \{(pn, bw, dir) \in P : dir = e\}$ und entsprechend **Ausgabeports** P_a. Außerdem gehört zu jedem Port $p = (pn, bw, dir)$ die Menge seiner **Signalwerte** $SI_p = \{0, ..., 2^{bw} - 1\}$.

Die Peripherie ist mit dem Kommunikationsautomat verträglich durch die **Input-** und **Output-Strukturbedingungen** $I = \bigtimes_{p \in P_e} SI_p$ und $O = \bigtimes_{p \in P_a} SI_p$. Ein Input-Ablauf $(i_1, ...)$ mit zugehörigem Output-Ablauf $(o_1, ...)$ induziert für jeden Ein- bzw. Ausgabeport p einen **zugehörigen Signal-Ablauf** $(proj_p(i_1), ...)$ bzw. $(proj_p(o_1), ...)$.

In einer wie oben definierten Kommunikationsstruktur können die Indizes 1,2,... eines Ablaufes als **Takte** einer RTL-Implementierung interpretiert werden; es existiert also nur ein Takt für eine derartige Kommunikationsstruktur. Wie mit diesem Ansatz auch Systeme mit mehreren Taktdomänen erfasst werden können, erläutert Anhang C.

Definition 3.2 :
Für eine **Input-Abstraktion** $J \subseteq I^*$, ist ein Algorithmus A eine **taktgenaue J-Simulation** einer Kommunikationsstruktur, falls er für jeden Input-Ablauf $\overline{i} \in J$ und jeden Port p den zugehörigen Signal-Ablauf berechnet. Für $J = I^*$ liegt eine **taktgenaue Simulation** vor.

Eine taktgenaue Simulation existiert offenbar in Form einer RTL-Simulation. Eine taktgenaue J-Simulation simuliert eine Kommunikationsstruktur nur für bestimmte Abläufe von Inputs korrekt. Für Abläufe, die nicht in J enthalten sind, macht die Simulation keine Aussage.

Definition 3.3 :
Zu einer Kommunikationsstruktur KS sei eine Überdeckung $PG \subset 2^P$ aller Ports durch **Portgruppen** $pg \in PG$ gegeben sowie eine (maximale) **Beobachtungsdauer** $n \in \mathbb{N}$. Eine **(PG-n-)Relevanzauswahl** ordnet jeder Portgruppe $pg \in PG$ eine **Relevanz** $R(pg) \subseteq I^n \times O^n$ zu.

Bei einer solchen Relevanzauswahl brauchen die Inputs und Outputs der Kommunikationsstruktur höchstens n Takte lang beobachtet zu werden. Danach steht fest, ob die Signalwerte der interessierenden Portgruppe am Ende der Beobachtung relevant sind. Sind sie relevant, werden die Output-Signalwerte der Portgruppe durch die nachfolgend definierte R-Simulation auch tatsächlich berechnet und stimmen an den relevanten Stellen mit den Abläufen in der simulierten Kommunikationsstruktur überein. Darüber hinaus lässt sich durch Abzählen der irrelevanten Takte die Taktgenauigkeit der Simulationsergebnisse aufrecht erhalten.

Definition 3.4 :
Zu einer Kommunikationsstruktur KS mit Relevanzauswahl R ist ein Algorithmus A eine **taktgenaue R-Simulation**, falls
$\forall$ Input-Abläufe $(i_1, ...)$ mit jeweils zugehörigem Output-Ablauf $(o_1, ...) = \overline{\lambda}((i_1, ...))$
$\forall$ Takte $t \in \mathbb{N}$ $(t \geq n)$
$\forall$ Portgruppen $pg \in PG$: $(i_{t-\nu}, o_{t-\nu})_{\nu=n-1,...,0} \in R(pg) \Rightarrow A$ berechnet $(proj_\pi(o_t))_{\pi \in pg \cap P_a}$

Definition 3.5 :
Analog zu Definition 3.4 definiert man eine **taktgenaue J-R-Simulation**.

In Defintion 3.4 wird deutlich, dass eine R-Simulation (und auch eine J-R-Simulation) erst ab einem Takt $t \geq n$ die Kommunikationsstruktur korrekt simuliert. Ohne Beschränkung der Allgemeinheit wird im weiteren Verlauf der Arbeit stets davon ausgegangen, dass $t \geq n$ gilt, da das endliche Teilstück der gesamten Simulation bis Takt n von einer taktgenauen Simulation (z.B. einer RTL-Simulation) berechnet werden kann und erst ab dann eine taktgenaue R-Simulation eingesetzt wird. Bei Simulationen mit $n \ll |i|$ kann der Einfluss der taktgenauen Simulation auf die Simulationsperformance vernachlässigt werden. Wie die Abschnitte 3.2 und 6.2 und die Anhänge B und I zeigen werden, gilt typischer Weise $n \leq 3$. Sinnvolle Systemsimulationen simulieren hingegen Hunderte bis Millionen von Takten, so dass $n \ll |i|$ erfüllt wird.

Zusammenfassung

Abbildung 3.6 auf der nächsten Seite präsentiert zusammenfassend die bis hierhin eingeführten Begriffe. Eine Kommunikationsstruktur berechnet für alle Input-Abläufe aus I^* einen zugehörigen Outputablauf. Dagegen berechnet die J-R-Simulation der Kommunikationsstruktur nur für eine gewisse Untermenge aller Input-Abläufe (die Menge J) eine zugehörigen Outputablauf.

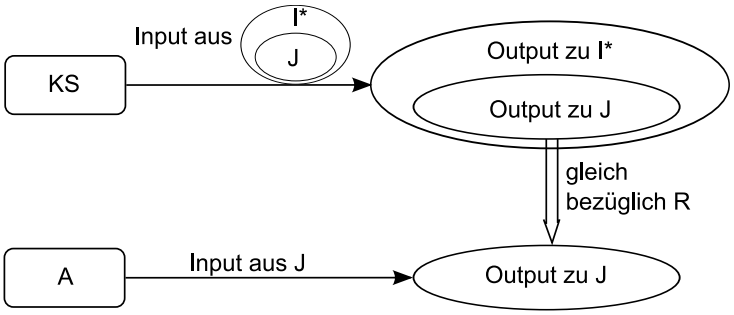


Abbildung 3.6.: Eine Kommunikationsstruktur KS und deren J-R-Simulation A

Die von der J-R-Simulation berechneten Outputabläufe müssen dann nicht mit den von der Kommunikationsstruktur berechneten Outputabläufen für J identisch sein. Die Übereinstimmung wird mit Hilfe von R auf spezielle Takte eines Outputablaufes reduziert („die relevanten Takte“) und in solchen relevanten Takten müssen nur die Werte der Ports übereinstimmen, die den Takt im von der Kommunikationsstruktur berechneten Outputablauf als relevant markierten; die Werte aller anderen Ports können beliebig sein. In nicht-relevanten Takten können die Werte aller Ports beliebig sein.

Eine J-R-Simulation operiert also im Vergleich zu der simulierten Kommunikationsstruktur auf einer reduzierten Inputmenge und produziert daher eine kleinere Outputmenge. Darüber hinaus sind die Anforderungen bezüglich der erwarteten Werte an die ohnehin kleinere Outputmenge im Vergleich zur Kommunikationsstruktur abgeschwächt. Diese beiden Eigenschaften erlauben es einer J-R-Simulation schneller zu simulieren als die Kommunikationsstruktur selbst. Dabei hängt das Ausmaß der potentiellen Geschwindigkeitssteigerung stark von der Definition von J und R ab. Aus diesem Grund wird das folgende Kapitel Anwendungsfälle von taktgenauem TLM untersuchen und dafür geeignete J- und R-Definitionen bestimmen.

3.2. Anwendungen der taktgenauen Simulation

In Abschnitt 3.1 wurde gezeigt, dass zur Definition einer taktgenauen J-R-Simulation die Input-Abstraktion J und die Relevanzauswahl R das Ausmaß der Abstraktion und der erzielbaren Simulationsaussagen festlegen. Ein Standard für taktgenaue TLM-Simulation muss also auch festlegen, wie J und R zu definieren sind. Die allgemeinen Definitionen 3.2 und 3.3 sind aber nicht leicht handhabbar. Aus diesem Grund werden nun die wichtigsten Anwendungsfälle von taktgenauem TLM untersucht und dafür anwendungsfallabhängige Definitionen festgelegt. Im weiteren Verlauf der Arbeit wird dann gezeigt, wie ein Standard für taktgenaue TLM-Simulation unter Verwendung dieser Definitionen aussehen kann.

Eingehende Literaturrecherchen und Diskussionen in verschiedenen Arbeitsgruppen (OSCI-TLM-Working-Group (WG), OCP-System-Level-Design (SLD)-WG, GreenSocs Corporation) zeigen die folgenden Anwendungsfälle als die wichtigsten für taktgenaues TLM auf.

3.2.1. Performance-Evaluation

Bei Performance-Tests bzw. **Performance-Evaluationen** [Aldi06, Aziz09, Donl04, Ghen05, LeYi07, ScLM06, WiHO06],[KoHA05]$_i$ soll die zu erwartende Kommunikationsperformance (Latenzen, Durchsätze) ermittelt werden. Dazu werden die Module des Systems oft als sogenannte Traffic-Generatoren und Datensenken modelliert, während die Kommunikationsstrukturen die Kommunikationslatenzen und -durchsätze taktgenau widerspiegeln. In einem solchen Aufbau spielen die übertragenen Daten nur dann eine Rolle, wenn Latenzen und Durchsätze datenabhängig sind. Entscheidend sind die Taktschritte, zu denen Kommunikation stattfindet. Darüber hinaus kann auch von Protokollverstößen innerhalb der Kommunikation abstrahiert werden. So kann z.B. vom expliziten Rücksetzen von Signalen abstrahiert werden, sodass fehlerhaftes Ansteuern der Signale nicht mehr modellierbar ist. Es müssen nur noch die Effekte der Signale modelliert werden, die Einfluss auf das Zeitverhalten der Kommunikation haben können, sodass eine starke Datenabstraktion möglich ist (sind Latenzen und Durchsätze nicht datenabhängig, kann vollständig von den Nutzdaten abstrahiert werden). Ziel der Performance-Evaluation ist es, extreme Lastsituationen zu identifizieren und zu analysieren, wie das System darauf reagiert. So können Flaschenhälse aufgefunden und so gegebenenfalls die Systemarchitektur angepasst werden. Ein taktgenaues Performance-Modell ist meist homogen in Hinsicht auf die Kommunikationsabstraktion, jedoch können IP-Blöcke, die nur sporadisch verwendet werden oder deren Performance-Aussagen auch mit Hilfe abstrakterer Modellierung genau genug sind, mit Hilfe der Mixed-Mode-Simulation ([SCP+03]) eingebunden werden.

Eine besondere Form der Performance-Evaluation ist die **Echtzeit-Anforderungsverifikation** [Ghen05, LeYi07, ScLM06][Engb08]$_i$. Dabei wird ein System bezüglich seiner zu erwartenden Antwortzeit auf spezielle Ereignisse hin untersucht. Dabei muss festgestellt werden, ob das System die festgelegten weichen oder harten Echtzeitanforderungen [Lapl04] unter verschiedenen Bedingungen einhält. Alle Systemmodule, die direkt an der Bearbeitung der eingehenden Ereignisse und an der Erzeugung der Systemantwort beteiligt sind, werden dabei taktgenau modelliert; Software wird auf taktgenauen Instruction-Set-Simulators (ISSs) ausgeführt. Module, die nicht unmittelbar beteiligt sind, können dabei sehr abstrakt modelliert werden. Bei der Verifikation von Echtzeitanforderungen ist es wünschenswert, das System zunächst schnell in einen speziellen Zustand zu versetzen (z.B. das Betriebssystem zu booten), bevor die eigentliche Analyse beginnt. Dazu müssten die Module umschaltbar zwischen taktgenauer und abstrakter Modellierung sein. Nachdem das System schnell mit Hilfe der abstrakten Modellierung den gewünschten Zustand erreicht hat, kann dann auf die taktgenaue Modellierung umgeschaltet werden. Ein Modell zur Verifikation von Echtzeitanforderungen ist in Hinsicht auf die Kommunikationsabstraktion oft heterogen.

Meine Untersuchungen verschiedener MMBIF und auch proprietärer IP-Interfaces zeigen, dass es zur exakten Bestimmung der Latenzen und Durchsätze einer Kommunikationsstruktur ausreicht, die Start- und Endtakte von Busphasen (Definition 3.7) zu bestimmen. Anhang B zeigt die Ergebnisse meiner Analyse von Busphasen auf. Eine taktgenaue J-R-Simulation

zum Zwecke der Performance-Evaluation muss diese Starts und Enden zu exakt den gleichen Takten berechnen wie eine vergleichbare taktgenaue Simulation (z.B. eine RTL-Simulation).

Definition 3.7 :
Gegeben seien eine Kommunikationsstruktur und daraus ein Master oder Slave $ms \in M \cup S$. $\Phi = (PP_{\Phi S}, PP_{\Phi E}, PP_{\Phi O}, tb_\Phi, S_\Phi, E_\Phi, A_\Phi)$ ist eine Busphase von ms, mit der Menge der **(Bus-)Phasenstartports** $PP_{\Phi S} \subseteq pz(ms)$, die beim Start der Phase berechnet bzw. gesetzt werden müssen, der Menge der **(Bus-)Phasenendports** $PP_{\Phi E} \subseteq pz(ms)$, die beim Ende der Phase berechnet bzw. gesetzt werden müssen, der **(Bus-)Phasenobservierungsports** $PP_{\Phi O} \subseteq pz(ms)$, deren Werte Start oder Ende der Phase beeinflussen, ohne dass die Phase selbst die Werte bestimmt, der **maximalen Busphasen-Beobachtungsdauer** tb_Φ und, mit $PP_\Phi = PP_{\Phi S} \cup PP_{\Phi E}$ als Menge der **(Bus-)Phasenports**, den **Start-**, **Ende-** und **Abbruchkriterien**

$$S_\Phi, E_\Phi, A_\Phi \subseteq \left(\bigtimes_{p \in (PP_\Phi \cup PP_{\Phi O}) \cap P_e} SI_p \times \bigtimes_{p \in (PP_\Phi \cup PP_{\Phi O}) \cap P_a} SI_p \right)^{tb_\Phi}$$

Φ_{ms} ist die **Busphasenmenge eines Teilnehmers** ms.

$\Phi_{Per} = \bigcup_{ms \in M \cup S} \Phi_{ms}$ ist die **Busphasenmenge der Peripherie**.

$tb_{\Phi_{Per}} = \max(\{tb_\Phi | \Phi \in \Phi_{Per}\})$ ist die **maximale Beobachtungsdauer der Peripherie**.

Wie in Anhang B erläutert sind die Signale einer Busphase zwischen Ende und Start ohne Bedeutung. Die Signale der Phase bleiben nach dem Start grundsätzlich bis zum Ende der Phase stabil. Eine Änderung ist nur zulässig, wenn dies die Phase beendet oder abbricht. Die Signale, die vom Empfänger der Phase getrieben werden, sind nur am Phasenende gültig und sind zu allen anderen Zeitpunkten nur insoweit bestimmt, dass sie anzeigen, dass die Phase nicht endet. Abbildung 3.8 verdeutlicht dies anhand einer Multitaktphase.

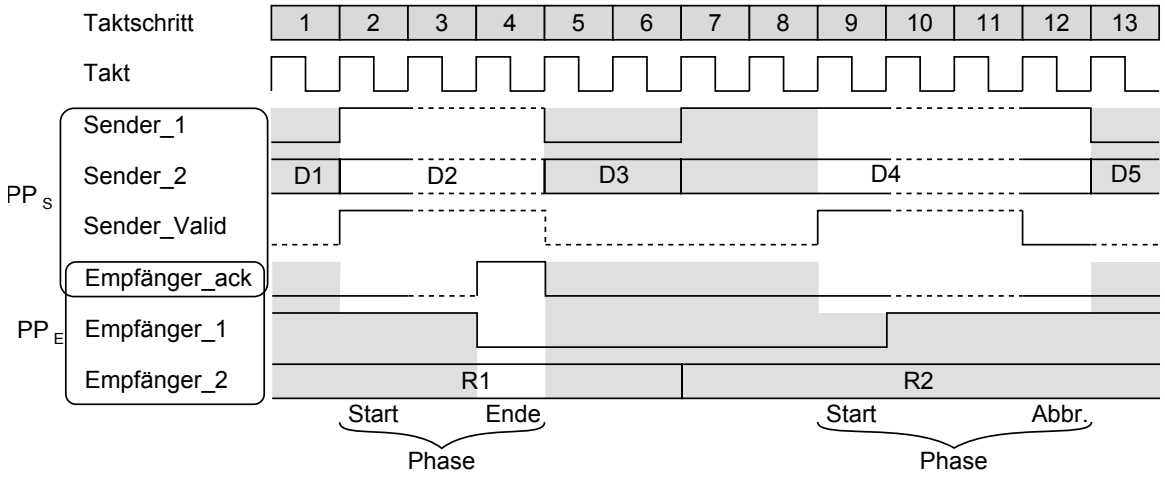


Abbildung 3.8.: Exemplarische Phasen: Bestätigt und abgebrochen

Durchgezogene, weiß hinterlegte Signalkurven deuten an, dass diese Signale explizit gesetzt bzw. berechnet werden müssen. Gestrichelte Signalkurven sind aufgrund der oben genannten

Phaseneigenschaften ableitbar. Grau hinterlegte Bereiche markieren Signale, deren Werte ohne Bedeutung sind. Man erkennt eine Phase, die in Takt 2 startet und in Takt 4 endet, und eine Phase, die in Takt 9 startet und in Takt 12 abgebrochen wird. Zum Phasenstart müssen alle Sendersignale und das Signal Empfänger_ack gesetzt bzw. berechnet werden, damit einerseits klar ist, welche Daten der Sender kommunizieren will und ob die Phase sofort endet. Beim Phasen-Ende müssen alle Empfänger-Signale berechnet bzw. gesetzt werden, damit klar ist, auf welche Art und mit welcher Antwort die Phase endet. Bei einem Abbruch werden die gleichen Signale wie beim Start gesetzt bzw. berechnet.

Der Verlauf der Kommunikation in Abbildung 3.8 kann als Repräsentant aller Abläufe angesehen werden, die in den nicht grau hinterlegten Bereichen identisch sind. Nun definiere ich die Input-Abstraktion J zur Performance-Evaluation derart, dass sie für jede dieser Klassen von Input-Abläufen nur noch genau einen Repräsentanten enthält.

Für jeden Port $p \in P$ der Peripherie einer Kommunikationsstruktur wird ein **Default-Zustand** $dft(p) \in SI_p$ festgelegt. Für Phasen mit $E_\Phi \cup A_\Phi \neq \emptyset$ legt dieser den Wert der Phasenports nach dem Ende oder Abbruch und vor dem Start (also außerhalb) einer Phase fest und muss so definiert sein, dass die Phase als nicht startend und nicht endend identifizierbar ist. Für Phasen mit $E_\Phi \cup A_\Phi = \emptyset$ ist dieser Wert beliebig aber fix. Es handelt sich um den Initialwert der Phasenports. Abschnitt B.2.5 erläutert, wie für die von mir identifizierten Busphasentypen Default-Zustände gefunden werden können.

Definition 3.9 :
Zu einer Kommunikationsstruktur sei die Menge aller Phasen der Peripherie Φ_{Per} gegeben. Die Funktion zur Berechnung der **Phasen-Historie** *hist* aus einem Input-Ablauf $\overline{i} \in I^*$ und einem Output-Ablauf $\overline{o} \in O^*$ für eine Phase $\Phi \in \Phi_{Per}$ und einen Takt $t \in \mathbb{N}$ mit $|\overline{i}| = |\overline{o}|$ und $tb_\Phi \leq t \leq |i|$ ergibt sich dann zu:

$$hist : (\Phi, \overline{i}, \overline{o}, t) \mapsto ((proj_\pi(i_\nu))_{\pi \in PP_\Phi \cap P_e}(proj_\pi(o_\nu))_{\pi \in PP_\Phi \cap P_a})_{\nu = t - tb_\Phi + 1, \ldots, t} \quad (1)$$

Für $\overline{o} = \overline{\lambda}(\overline{i})$ bestimmt sich die Funktion zur Berechnung der **Input-abhängigen Phasen-Historie** $hist_\lambda$ als

$$hist_\lambda : (\Phi, \overline{i}, t) \mapsto hist((\Phi, \overline{i}, \overline{\lambda}(\overline{i}), t)) \quad (2)$$

Mit Hilfe der Input-abhängigen Phasen-Historie kann die Input-Abstraktion J zur Performance-Evaluation definiert werden. Dazu werden die Mengen *beg*, *end*, *abr* und *def* wie in Definition 3.10, Gleichungen 1 bis 4 verwendet. Ein Element $(p, t, \overline{i})$ ist genau dann in *beg*, wenn eine Phase, zu der Port p gehört, in Takt t im Input-Ablauf $\overline{i}$ (mit zugehörigem Output-Ablauf) startet und p beim Start der Phase gesetzt bzw. berechnet werden muss. Analog bestimmen sich *end* und *abr* für das Ende bzw. den Abbruch einer Phase. Ein Element $(p, t, \overline{i})$ ist genau dann Element von *def*, wenn eine Port p zu einer Phase gehört, die im Takt $t - 1$ endete oder abbrach.

Definition 3.10 :
Gegeben seien eine Kommunikationsstruktur, die Menge aller Phasen der Peripherie Φ_{Per} sowie die Funktion zur Berechnung der Input-abhängigen Historie $hist_\lambda$.
Es definieren sich die Mengen *beg*, *end*, *abr* und *def* wie folgt:

$$beg = \left\{ (p,t,\bar{i}) \in P_e \times \mathbb{N} \times I^* \;\middle|\; \exists \Phi \in \Phi_{Per} : p \in PP_{\Phi S} \wedge hist_\lambda((\Phi,\bar{i},t)) \in S_\Phi \right\} \quad (1)$$

$$end = \left\{ (p,t,\bar{i}) \in P_e \times \mathbb{N} \times I^* \;\middle|\; \exists \Phi \in \Phi_{Per} : p \in PP_{\Phi E} \wedge hist_\lambda((\Phi,\bar{i},t)) \in E_\Phi \right\} \quad (2)$$

$$abr = \left\{ (p,t,\bar{i}) \in P_e \times \mathbb{N} \times I^* \;\middle|\; \exists \Phi \in \Phi_{Per} : p \in PP_{\Phi S} \wedge hist_\lambda((\Phi,\bar{i},t)) \in A_\Phi \right\} \quad (3)$$

$$def = \left\{ (p,t,\bar{i}) \in P_e \times \mathbb{N} \times I^* \;\middle|\; \exists \Phi \in \Phi_{Per} : p \in PP_{\Phi} \wedge hist_\lambda((\Phi,\bar{i},t-1)) \in E_\Phi \cup A_\Phi \right\} \quad (4)$$

Damit ist die **Input-Abstraktion zur Performance-Evaluation** gegeben durch:

$$J = \left\{ \bar{i} \in I^* \middle| \begin{array}{l} \forall t \in \mathbb{N}, tb_{\Phi_{Per}} < t \leq |\bar{i}|, \forall p \in P_e : \\ \quad proj_p(i_t) \neq proj_p(i_{t-1}) \Rightarrow (p,t,\bar{i}) \in beg \cup end \cup abr \cup def \\ \quad \wedge \quad (p,t,\bar{i}) \in def \backslash (beg \cup end \cup abr) \Rightarrow proj_p(i_t) = dft(p) \end{array} \right\} \quad (5)$$

Somit enthält nun J (Definition 3.10, Gleichung 5) nur Inputabläufe, bei denen sich ab einem Takt $t > tb_{\Phi_{Per}}$ ein Port nur dann ändern darf, wenn eine zugehörige Phase startet, endet oder abbricht oder wenn im vorangegangenen Takt die Phase endete oder abbrach. Ist letzteres der Fall und darf sich der Port einzig deswegen ändern, so muss er in den Default-Zustand wechseln. In allen Takten mit $t < tb_{\Phi_{Per}}$ ist J beliebig. Im Folgenden wird (ähnlich wie schon bei der R-Simulation erwähnt) ohne Beschränkung der Allgemeinheit dieses endliche Anfangsstück von J nicht betrachtet, da dafür eine taktgenaue Simulation (z.B. RTL) eingesetzt werden kann und erst ab dann die J-Simulation zur Performance-Evaluation verwendet wird.

Um noch einmal das Beispiel aus Abbildung 3.8 auf Seite 28 aufzugreifen: Die Signalwerte in den grau hinterlegten „Don't-care-Bereichen“ werden in J stets auf den Default-Zustand festgelegt. In Takten mit durchgezogenen, weiß hinterlegten Signalverläufen sind Änderungen in J zulässig. Die gestrichelten Verläufe erzwingt J, da es zu diesen Takten grundsätzlich keine Änderung erlaubt. Die gestrichelte Änderung in Takt 5 beim Signal Sender_Valid ist ein Wechsel in den Default-Zustand aufgrund des Phasenendes in Takt 4. Diese Änderung ist gestrichelt, da sie aus der Tatsache, dass die Phase nicht startet, ableitbar ist.

Derart eingeschränkte Abläufe von Inputs können in der Simulation stark komprimiert werden. Es genügt, der J-Simulation lediglich die Signaländerung von Ports mitzuteilen, die sich an den **(Bus-)Phasenrändern**, also bei Start, Ende oder Abbruch einer Phase, ändern dürfen. Die Werte zu allen anderen Takten müssen nicht mitgeteilt werden, da diese aus der Input-Abstraktion zur Performance-Evaluation folgen. Dies kann die Anzahl der notwendigen Eingaben stark (siehe Anhang D) und somit auch die notwendige Aktivität in der Simulation reduzieren.

Es kann nicht immer davon ausgegangen werden, dass jeder Master und Slave einen J-konformen Input-Ablauf für die taktgenaue J-Simulation zur Performance-Evaluation erzeugen wird. Zum Beispiel kann für den Kommunikationsautomaten eine taktgenaue J-Simulation verwendet werden, während die Peripherie ohne Input-Abstraktion simuliert wird[2]. Unter der Annahme, dass die Master und Slaves zumindest nicht gegen die grundlegenden Phaseneigenschaften verstoßen (siehe Abschnitt B.2), muss also eine Möglichkeit gefunden werden, die Abläufe aus I^*, die in den gültigen Bereichen übereinstimmen, während der Simulation auf ihren Repräsentanten aus J abzubilden.

Definition 3.11 :
Gegeben sei eine Kommunikationsstruktur und die Menge aller Busphasen der Peripherie Φ_{Per}.
Für eine Portgruppe $pg \in 2^P$, einen Port $p \in P$ und zwei Signalwerte $opt1, opt2 \in SI_p$ ergibt sich die **Latch-Funktion** g zu:

$$g : (p, pg, opt1, opt2) \mapsto \begin{cases} opt1 & \text{falls } p \in pg \\ opt2 & \text{sonst} \end{cases} \tag{1}$$

Für eine Phase $\Phi \in \Phi_{Per}$, einen Input-Ablauf $\overline{i} \in I^*$ und einen Output-Ablauf $\overline{o} \in O^*$ mit $|\overline{i}| = |\overline{o}|$, berechnet die **($I^*$-$J$-)Filter-Funktion** $filter$ für Takt $t \in \mathbb{N}$ die Werte der Eingabeports der Phase:

$$filter : (\Phi, \overline{i}, \overline{o}, t) \mapsto \begin{cases} (proj_\pi(i_t))_{\pi \in PP_\Phi \cap P_e} & \text{falls } t < tb_\Phi \\ \hline (g(\pi, PP_{\Phi_S}, proj_\pi(i_t), dft(\pi)))_{\pi \in PP_\Phi \cap P_e} & \text{falls } t \geq tb_\Phi \wedge hist(\Phi, \overline{i}, \overline{o}, t) \in (S_\Phi \cup A_\Phi) \backslash E_\Phi \\ \hline (g(\pi, PP_{\Phi_E}, proj_\pi(i_t), proj_\pi(filter(\Phi, \overline{i}, \overline{o}, t-1))))_{\pi \in PP_\Phi \cap P_e} & \text{falls } t \geq tb_\Phi \wedge hist(\Phi, \overline{i}, \overline{o}, t) \in E_\Phi \backslash (S_\Phi \cup A_\Phi) \\ \hline (proj_\pi(i_t))_{\pi \in PP_\Phi \cap P_e} & \text{falls } t \geq tb_\Phi \wedge hist(\Phi, \overline{i}, \overline{o}, t) \in (S_\Phi \cup A_\Phi) \cap E_\Phi \\ \hline (dft(\pi))_{\pi \in PP_\Phi \cap P_e} & \text{falls } t \geq tb_\Phi \wedge hist(\Phi, \overline{i}, \overline{o}, t) \notin S_\Phi \cup E_\Phi \cup A_\Phi \wedge hist(\Phi, \overline{i}, \overline{o}, t-1) \in E_\Phi \cup A_\Phi \\ \hline filter(\Phi, \overline{i}, \overline{o}, t-1) & \text{sonst} \end{cases} \tag{2}$$

[2]So könnte z.B. das in Abschnitt 6.3 beschriebene J-R-TLM-Modell des PLB mit normalen RTL-Mastern oder Slaves betrieben werden.

Die Filterfunktion $filter$ (Definition 3.11, Gleichung 2) sorgt dafür, dass für alle Takte $t < tb_\Phi$ alle Input-Änderungen übernommen werden[3]. Danach werden Änderungen von Eingabeports nur übernommen, wenn diese mit Rändern von Phasen zusammenfallen und dies auch nur für Ports, die beim entsprechenden Phasenrand gesetzt werden müssen. Startet eine Phase, ohne im gleichen Takt zu enden, so werden alle Ports, die nicht bei Start gesetzt oder berechnet werden müssen, in den Default-Zustand gezwungen. In Takten, die auf Takte folgen, in denen eine Phase endete oder abbrach und in denen die Phase nicht erneut startet, wird der Default-Zustand angenommen. In allen anderen Fällen bleiben die Werte unverändert.

Die Filterfunktion benötigt zum Filtern in Takt t einen $tb_\Phi + 1$ Takte langen Rückblick in die Vergangenheit der Phasenportabläufe (aufgrund von $hist(\Phi, \bar{i}, \bar{o}, t-1)$) und den einen Takt langen Rückblick auf den Ablauf der eigenen Filterergebnisse (aufgrund von $filter(\Phi, \bar{i}, \bar{o}, t-1)$). Da die Filterung für Takt t keinen Zugriff auf Elemente von $\bar{i}$ oder $\bar{o}$ mit einem Index größer als t benötigt, genügt es $\bar{i}$ und $\bar{o}$ bis einschließlich Takt t zu kennen. Die Filterung kann also zur Simulationslaufzeit während der Simulation von Takt t erfolgen. Dazu benötigt eine Implementierung von $filter$ lediglich einen tb_Φ langen Puffer zum Speichern der vergangenen Werte (zusammen mit dem aktuellen Wert ergibt sich der $tb_\Phi + 1$ lange Rückblick). Der einen Takt lange Rückblick auf die Ergebnisse ist schlicht durch den aktuellen Zustand der Filterung gegeben[4].

Hat man $filter$ für jede Phase implementiert, so kann aus den Filterergebnissen wieder ein vollständiger Input $j \in J$ derart zusammengesetzt werden, dass sich für Ports p, die zu mehr als einer Phase gehören, der vom Default-Zustand verschiedene Wert durchsetzt. Gehört ein Port p nur zu einer einzigen Phase, ist seine Integration in den Input $j \in J$ trivial.

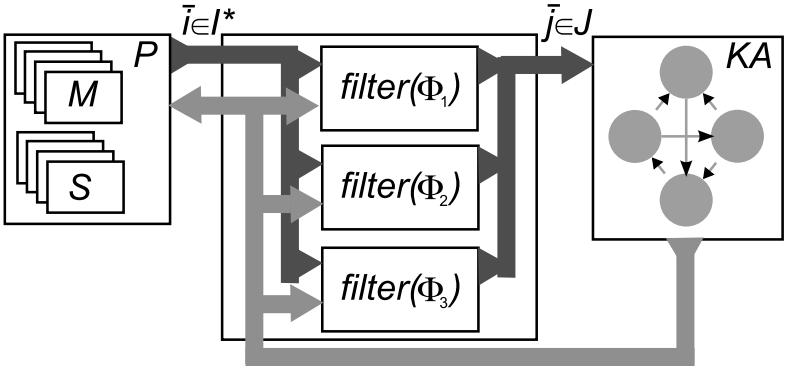


Abbildung 3.12.: Input-Filterung zur Performance-Evaluation

Die Verwendung solcher Filter ist in Abbildung 3.12 skizziert. Das J-Simulationsmodell des Kommunikationsautomaten kann dann aus der J-konformen Ansteuerung großen Nutzen

[3]Es sei nocheinmal darauf hingewiesen, dass für dieses endliche Teilstück eine taktgenaue Simulation eingesetzt wird und noch keine J-Simulation

[4]Dieser Ansatz kann dann in den TLM-RTL-Adaptern verwendet werden, die im in Fußnote [2] erwähnten Falll benötigt werden.

ziehen. Zum Beispiel kann es rein reaktiv arbeiten[5]. Das bedeutet, es ist nicht notwendig, zu jedem Takt die Inputs abzufragen. Es kann ausschließlich auf Input-Änderungen reagiert werden, da diese nur an den Phasenrändern auftreten.

Die Relevanzauswahl R für eine Kommunikationsstruktur kann auch mit Hilfe der Busphasen beschrieben werden.

Die Überdeckung der Ports für die Relevanzauswahl ist die Zerlegung der Ports der Peripherie bezüglich ihrer Phasenstart- und -endzugehörigkeit (Definition 3.13, Gleichung 1). Die Beobachtungsdauer für die Relevanzauswahl ist die maximale Beobachtungsdauer der Peripherie (Definition 3.13, Gleichung 2). Relevant sind nur Takte, in denen Phasenränder auftreten (Definition 3.13, Gleichung 3).

Definition 3.13 :
Gegeben sei eine Kommunikationsstruktur und die Busphasenmenge der Peripherie Φ_{Per}. Dann ergibts sich die **Relevanzauswahl zur Performance-Evaluation** zu:

$$PG = \{PP_{\Phi S} | \Phi \in \Phi_{Per}\} \cup \{PP_{\Phi E} | \Phi \in \Phi_{Per}\} \tag{1}$$

$$n = tb_{\Phi_{Per}} \tag{2}$$

$$R(pg) := \left\{ (\bar{i}, \bar{o}) \in I^n \times O^n | \begin{array}{l} \exists \Phi \in \Phi_{Per} : \\ (pg = PP_{\Phi S} \wedge hist(\Phi, \bar{i}, \bar{o}, n) \in S_\Phi \cup A_\Phi) \\ \vee (pg = PP_{\Phi E} \wedge hist(\Phi, \bar{i}, \bar{o}, n) \in E_\Phi) \end{array} \right\} \tag{3}$$

Eine taktgenaue R-Simulation wird also wie gefordert in einem Takt, in dem Busphasen an den Schnittstellen der Master und Slaves starten, abbrechen oder enden, die Signalwerte der Ausgabeports berechnen, die beim entsprechenden Phasenrand gesetzt werden müssen. Die Signalwerte aller anderen Ausgabeports werden nicht berechnet. Die Werte der nicht berechneten Ausgabeports werden wie folgt festgelegt:

Zwischen einem Ende oder Abbruch und einem Start nehmen sie den Default-Zustand an. Die an einem Phasenrand nicht berechneten Ausgabeports der Phase behalten ihren letzten Wert.

Zusammenfassend kann eine taktgenaue J-R-Simulation zur Performance-Evaluation taktgenau die Phasenränder berechnen, indem irrelevante Takte gezählt werden und bei relevanten Takten die Phasenports für den jeweiligen Phasenrand berechnet werden. Für irrelevante Takte, oder für relevante Takte, in denen nicht alle existierenden Phasen enden oder starten, werden die Phasenports der nicht startenden bzw. nicht endenden Phasen ohne Berechnung festgelegt: Zwischen Start und Ende einer Phase gibt es keine Änderungen, zwischen Ende und Start einer Phase wird der Default-Zustand angenommen. Daher liegt eine Use-Case-Abstraktion vor: Es ist unmöglich, die (illegale) Änderung von Phasenports zwischen Phasenstart und -ende zu modellieren. Das heißt, es wird von der Modellierung

[5]Wie z.B. das in Abschnitt 6.3 beschriebene Modell.

von Protokollverstößen abstrahiert, der Use-Case der Protokollkonformitätsverifikation innerhalb einer Phase entfällt. Das Entfernen von Ports ist eine Art der Datenabstraktion, kann aber auch zu einer Strukturabstraktion führen, wenn z.B. der gesamte Datenpfad der Kommunikationsstruktur entfällt, da alle Nutzdatensignale ignoriert werden. Die Einschränkung der Inputs auf J ermöglicht oft eine Algorithmenabstraktion innerhalb der simulierten Kommunikationsstruktur. Das Ausmaß dieser Abstraktionen ist mit Hilfe der Definitionen der Busphasen(-ports) steuerbar.

3.2.2. Power-Analyse

Bei der **Power-Analyse** [CCC+03, DBD+06, VDK+08, ZRRJ04] wird ein System meist homogen in Hinsicht auf die Kommunikationsabstraktion modelliert. Dabei werden alle Module taktgenau simuliert (Software wird auf taktgenauen ISSs ausgeführt). Da Änderungen auf Datenleitungen auch relevant für die Power-Analyse sind, kann von diesen nicht abstrahiert werden. Darüber hinaus müssen im Gegensatz zur Performance-Evaluation und Echtzeit-Anforderungsanalyse alle Power-relevanten Signalwechsel explizit modelliert werden. Das bedeutet, dass in einen Modell zur Power-Analyse mehr IMCs auftreten als in Modellen zur Performance-Evaluation oder zur Echtzeit-Anforderungsanalyse, bei denen sich Signale nur an Phasenrändern ändern. Die mögliche Datenabstraktion ist minimal. Ähnlich zur Echtzeit-Anforderungsanalyse ist es hilfreich, das System schnell in einen speziellen Zustand zu versetzen. Dementsprechend ist ein Umschalten von abstrakter zu taktgenauer Modellierung zur Laufzeit auch hier wünschenswert. Dabei wird im Gegensatz zur Echtzeitanforderungsanalyse aber das gesamte System umgeschaltet. Ein Power-Analyse-Modell kann oft auch zur Performance-Evaluation und Echtzeitanforderungsanalyse eingesetzt werden, ist aber aufgrund seiner geringeren Abstraktion nicht so performant wie ein dediziertes Modell für Performance-Evaluation oder Echtzeitanforderungsverifikation.

Die exakte Definition einer J-R-Simulation zur Power-Analyse hängt von der angestrebten Genauigkeit der Power-Analyse ab. Die präziseste Aussage ist natürlich mit $J = I^*$ und einer Relevanzauswahl zu erzielen, die jeden Takt, in dem sich ein Signal ändert, als relevant markiert. Das entspricht dann aber einer RTL-Simulation.

Ein Weg ist eine geeignete Erweiterung einer J-R-Simulation zur Performance-Evaluation. Dazu definiert man für jeden Master und Slave $ms \in M \cup S$ der Peripherie die **Eingabe-Powerphase** pow_E, deren Phasenports eine Auswahl der ms zugeordneten Eingabeports der Kommunikationsstruktur sind und die **Ausgabe-Powerphase** pow_A, deren Phasenports einer Auswahl der ms zugeordneten Ausgabeports der Kommunikationsstruktur sind, jeweils als infinite Phase (siehe Anhang B).

Nimmt man die Ein- und die Ausgabe-Powerphase in die Menge der Busphasen auf, so erweitert sich J (Definition 3.10) so, dass sich alle Phasenports der Eingabe-Powerphase zu jedem beliebigen Zeitpunkt ändern dürfen, da eine Änderung stets einen (Neu-)Start der Phase bedeutet. Durch R (Definition 3.13, Gleichung 3) werden dann alle Taktschritte relevant, in denen sich mindestens ein Werte der Phasenports der Ausgabepowerphase ändert.

Eine J-R-Simulation zur Power-Analyse kann nun, wann immer eine Eingabe-Powerphase startet, die konsumierte Power berechnen und akkumulieren, während sie für jedes Eintreten von Power-relevanten Outputänderungen den Taktschritt und die Änderung berechnen muss.

3.2.3. Zusammenfassung

Die vorangegangenen Abschnitte haben verdeutlicht, dass taktgenaue busphasenbasierte J-R-Simulationen für die wichtigsten Anwendungsfälle von taktgenauem TLM, der Performance-Evaluation und der Power-Analyse, eingesetzt werden können. Anhang D zeigt, dass eine solche taktgenaue J-R-Simulation zur Performance-Evaluation oder Power-Analyse in der Regel weniger Inputs und Outputs erzeugen bzw. konsumieren muss als eine RTL-Simulation, also effizienter simuliert werden kann. Die busphasenbasierte Simulation ist aber nicht auf Performance- und Power-Analyse eingeschränkt. Sie ist für jeden Anwendungsfall geeignet, für den sich Busphasen bzw. Kommunikationsphasen nach Definiton 3.7 beschreiben lassen. Man kann damit z.B. auch RTL-Simulationen beschreiben, wenn man für jedes Signal eines jeden Interfaces eine infinite Busphase definiert.

3.3. TLM-2.0 für taktgenaue TLM

Meine Arbeit setzt sich mit der Bestimmung eines effizienten Interfaces im Rahmen der taktgenauen busphasenbasierten J-R-Simulation auseinander. Ein solches Interface muss die durch J und R gegebenen Einschränkungen ausnutzen, um größtmögliche Vorteile für die Simulationseffizienz zu erlauben. Da sich Eingabeports in der taktgenauen busphasenbasierten J-R-Simulation nur an Phasenrändern ändern können und nur dann relevante Takte in Outputabläufe auftreten, liegt ein Interface nahe, dass es erlaubt, innerhalb eines einzigen Vorganges alle sich ändernden Inputs bzw. Outputs zu übermitteln und dabei auch gleich den entsprechenden Phasenrand zu identifizieren. Es bietet sich also an, in der Simulation einen Funktionsaufruf zu verwenden, der zwei Objekte übermittelt. Eines, das die Input und Outputwerte enthält und eines, das die Phase bestimmt. Für Interfaces, die solche komplexen Objekte übermitteln, also auch für die taktgenaue busphasenbasierte J-R-Simulation, ist TLM sehr gut geeignet[6].

In Abschnitt 2.3 wurde aufgezeigt, dass mit TLM-2.0 ein TLM-Standard für abstrakte Kommunikationsmodellierung geschaffen wurde. Das Interface für taktgenaue busphasenbasierte J-R-Simulation kann nun entweder völlig davon entkoppelt sein oder aber auf dem Standard für abstraktere TLM-Modellierung aufsetzen. Eine von TLM-2.0 losgelöste Implementierung oder völlig eigenständige Interfacedefinitionen für verschiedene MMBIF, wie sie bereits existieren (z.B. [LoTs09], $[\text{ARM-07, IBM-06, OCP-08}]_i$), führten zu einem „Ausbruch" aus der Modellierung eines IP-Blocks auf verschiedenen Kommunikationsabstraktionen, wie in Abbildung 3.14 (heterogenes Schema) angedeutet.

[6]Dies wird auch noch einmal genauer in Abschnitt 4.2 erläutert.

Wie in Abschnitt 3.2 erläutert, ist die Mixed-Mode-Simulation ein wichtiger Faktor. Sind nun IP-Blöcke auf der taktgenauen Ebene vom Standard der abstrakteren Kommunikationsmodellierung entkoppelt (Abbildung 3.14 „heterogenes Schema“), werden für die Mixed-Mode-Simulation komplexe Adapter notwendig.

Darüber hinaus wurde in Abschnitt 3.2 auch aufgezeigt, dass ein IP-Block unter Umständen zur Simulationslaufzeit seine Kommunikationsabstraktion umschalten soll. Sind der abstrakte Standard und das Interface für taktgenaue busphasenbasierte *J-R*-Simulation entkoppelt, benötigt ein IP-Block zwei Schnittstellen (eine für TLM-2.0 und eine für die *J-R*-Simulation) und muss ein komplettes Sub-System des Modells doppelt existieren.

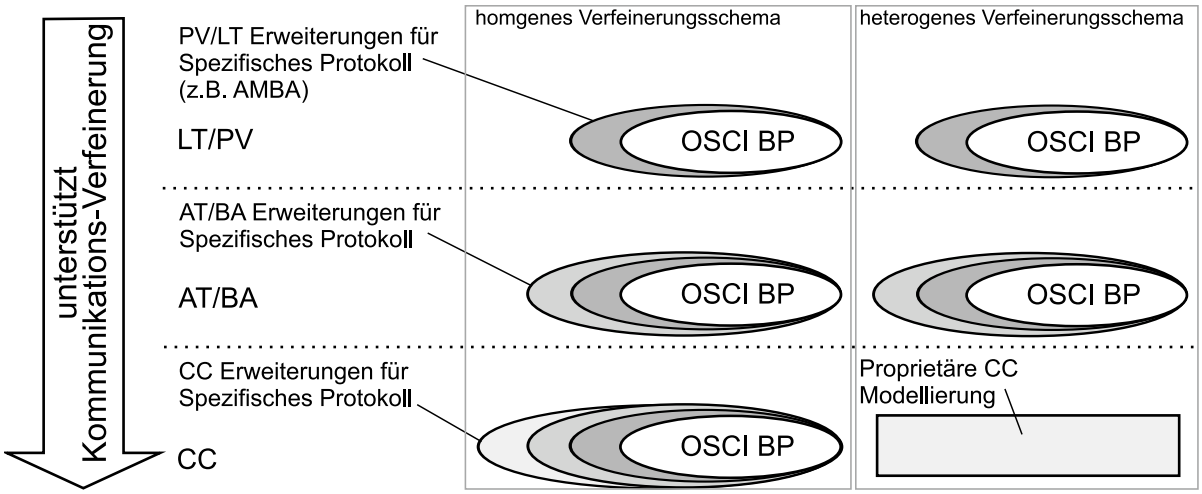


Abbildung 3.14.: Homogenität/Heterogenität im Modellierungsvorgehen

Existierte ein Interface für taktgenaue busphasenbasierte *J-R*-Simulation, der auf dem Standard für die abstraktere Kommunikationsmodellierung aufsetzte (Abbildung 3.14, homogenes Schema), würden die Adapter zwischen den Kommunikatonsabstraktionen vereinfacht und die Umschaltung dazwischen erforderte nur eine einzige Schnittstelle, über die dann entweder taktgenau oder abstrakter kommuniziert werden kann. Dies ist meiner Meinung nach ein noch wichtigerer Vorteil als die unten genannte Interoperabilität zwischen verschiedenen MMBIF.

Als weitere Vorteile können die gleichen Werkzeuge wie Transaktions-Logger, auf allen Ebenen des Entwurfs eingesetzt werden und Adapter zwischen verschiedenen MMBIF vereinfacht werden. Zusätzlich werden die Trainingszeiten zum Erlernen der taktgenauen Modellierung reduziert, da große Übereinstimmungen zur Modellierung auf höheren Kommunikatonsabstraktionsebenen existieren.

3.4. Fazit

Die Abschnitte 3.1 und 3.2 haben gezeigt, wie mit Hilfe eines Satzes von Busphasen die Inputabstraktion J und das Relevanzkriterium R für eine taktgenaue busphasenbasierte J-R-Simulation einer Kommunikationsstruktur definiert werden.

Damit ist nun klar, was die J-R-Simulation A in Abbildung 3.6 auf Seite 26 für einen gegebenen Satz Busphasen leisten muss. Es ist aber unklar, wie für eine gegebene Kommunikationsstruktur diese Simulation A konstruiert werden kann. Abbildung 3.15 zeigt das dafür von mir vorgeschlagene Vorgehen.

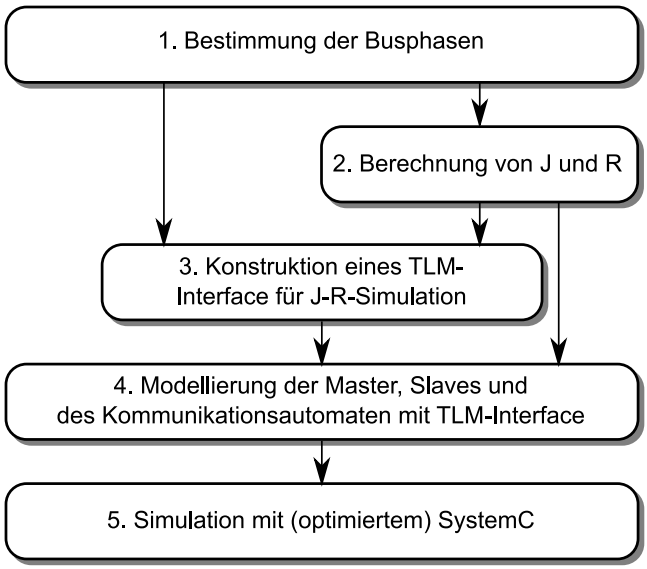


Abbildung 3.15.: Konstruktion einer taktgenauen busphasenbasierten J-R-Simulation mit TLM-2.0

Zuerst müssen die Busphasen bestimmt werden. Dieser Vorgang kann nicht automatisiert werden, jedoch ist es oft möglich, die in Anhang B erläuterten „Standard“-Phasen zu nutzen oder aber diese leicht verändert einzusetzen. Abschnitt 6.2 erläutert dieses Vorgehen an einem komplexen Beispiel.

Sind die Busphasen bestimmt, ergeben sich daraus algorithmisch J und R (Definitionen 3.10 und 3.13), sodass klar ist, welche Ereignisse (Busphasenstarts, -enden und -abbrüche) taktgenau bearbeitet werden und von welchen Signalen zu welchen Zeitpunkten abstrahiert wird. Basierend auf der Kenntnis von den Busphasen, der Inputabstraktion J und der Relevanzauswahl R wird dann ein TLM-Interface konstruiert, das auf die Verwendung für taktgenaue busphasenbasierte J-R-Simulation ausgelegt ist. Diese will ich auf TLM-2.0 aufbauen (siehe Abschnitt 3.3) und basierend darauf die TLM-Interfaces für die taktgenaue busphasenbasierte J-R-Simulation konstruieren.

Mit Hilfe dieses Interfaces und der Kenntnis, welche Ereignisse taktgenau stattfinden, können dann die Master, Slaves und der Kommunikationsautomat modelliert werden. Mit diesen Modellen und einem gegebenenfalls für die taktgenaue busphasenbasierte J-R-Simulation

optimiertem Simulator kann dann eine taktgenaue busphasenbasierte J-R-Simulation mit TLM-2.0 durchgeführt werden, welche deutlich schneller simuliert als eine entsprechende RTL-Simulation.

Damit ergeben sich die Ziele dieser Arbeit zu:

Z1 **Aufzeigen der grundsätzlichen Eignung von TLM-2.0 für taktgenaue busphasenbasierte J-R-Simulationen.**

Z2 **Festlegung eines TLM-2.0-Modellierungsstils für taktgenaue busphasenbasierte J-R-Simulationen.**

Z3 **Simulatorerweiterungen für taktgenaue busphasenbasierte TLM.**

Z4 **Erläuterung der Verwendung des Modellierungsstils an einem respräsentativen Beispiel und Aufzeigen des erzielbaren Speed-Ups.**

4. Taktgenaue busphasenbasierte J-R-Simulation mit TLM-2.0

Inhalt

4.1. Einleitung

In Abschnitt 3.2.1 wurde gezeigt, dass eine taktgenaue busphasenbasierte J-R-Simulation in der Regel weniger Inputs konsumieren und weniger Outputs generieren muss, um zu den gleichen Performance-Aussagen zu gelangen wie eine taktgenaue Simulation. Dies wird in Anhang D anhand des On-Chip-Peripheral-Bus (OPB) illustriert. In diesem Abschnitt wird gezeigt, dass sich TLM-2.0 besonders für solche J-R-Simulationen eignet. Die Abschnitte 4.2 bis 4.4 erfüllen Zielstellung Z1 aus Abschnitt 3.4 (Eignung von TLM-2.0 für taktgenaue busphasenbasierte J-R-Simulationen), während die Abschnitte 4.5 bis 4.11 die Zielstellung Z2 (Festlegung des TLM-2.0-Modellierungsstils für taktgenaue busphasenbasierte J-R-Simulationen) erfüllen.

4.2. Generic Payload und TLM-Phase

Ein busphasenbasiertes J-R-Modell benötigt in einem Takt als Input die Werte der Ports, die sich aufgrund eines Auftretens eines Phasenrandes geändern haben. Die Änderung auf

Default-Zustände muss dem Modell nicht übergeben werden, da sich diese Änderung aus den Phasenenden oder -abbrüchen im vorangegangenen Takt ergeben. Durfte sich kein Port ändern, ist keine Information notwendig.

Ein Master oder Slave der Peripherie kann dem Kommunikationsautomaten eine Änderung der ihm zugeordneten Ports mit Hilfe eines Interface-Method-Calls (IMCs) mitteilen, der einen Datencontainer mit den neuen Inputs übermittelt. Der Kommunikationsautomat kann dann den Datencontainer dahingehend analysieren, welche Phase startet, endet oder abbricht. Diese Analyse kann entfallen, wenn der Master oder Slave zu dem Datencontainer noch die Information hinzufügt, um welche Phase und welchen Phasenrand (Start, Ende, Abbruch) es sich handelt. Das Hinzufügen einer solchen Information ist effizienter als die Analyse im Kommunikationsautomaten, da der IMC entweder aus einem I^*-J-Filter kommt, und so die Analyse bereits erfolgt ist, oder aber aus einem Master, der bereits J-konform arbeitet und so mit dem Vorsatz kommuniziert, einen speziellen Phasenrand anzuzeigen. Dann ist das Hinzufügen der Information nahezu kostenfrei. Ein IMC gleicher Art kann vom Kommunikationsautomaten zur Übergabe der Outputs an Master oder Slaves verwendet werden.

Zur Veranschaulichung stellt Listing 4.1 einen IMC `announce` dar.

```
1  //fuer jedes Paar aus Phase und Phasenrand ein Wert im Aufzaehlungstyp
2  enum phase_event_id {
3    Phi1_Start, Phi1_End, Phi1_Abort,
4    Phi2_Start, Phi2_End, Phi2_Abort,
5    ...
6    Phin_Start, Phin_End, Phin_Abort,
7  };
8
9  //fuer jeden Port einen Wert
10 struct data_container
11 {
12   int p1, p2, p3, p4, ..., pm;
13 };
14
15 //ein IMC der neue Portwerte uebermittelt und den Phasenrand mitteilt
16 //  ports aus 'c', die sich geaendert haben koennen, sind abhaengig von 'ph' identifizierbar
17 void announce(data_container& c, phase_event_id& ph};
```

Listing 4.1: Ein Beispiel für einen IMC zur Übergabe von Input-/Output-Änderungen

Damit eine effiziente Übermittlung ohne Typkonvertierungen oder Kopien solcher Datencontainer vom Master zum Slave erfolgen kann, müssen Master und Slaves den gleichen Container verwenden, und da jeder Master potentiell mit jedem Slave kommunizieren kann, müssen alle Master und alle Slaves den gleichen Container-Typ verwenden.

Damit die Signatur des IMC unabhängig von der modellierten Kommunikationsstruktur ist, muss der Datencontainer so definiert sein, dass seine **Grundelemente**, also die Elemente, die fester Bestandteil des Containers sind, in nahezu jedem MMBIF existieren, und er muss erweiterbar sein, um auf das modellierte MMBIF angepasst zu werden.

Die gleichen Anforderungen können auch an den Datentyp gestellt werden, der die Phase und den Phasenrand bestimmt. Vergleicht man diese Anforderungen an den Datencontainer

und den Typ für den Phasenrand mit dem Generic Payload (GP) und der TLM-Phase aus dem Base-Protocol (BP) von TLM-2.0 (siehe Abschnitt 2.3.2) erkennt man, dass GP und TLM-Phase diese Anforderungen erfüllen. Die Verwendung von GP und TLM-Phase für taktgenaues TLM ergibt sich also nicht allein aus der Zielstellung, TLM-2.0 für taktgenaues TLM einzusetzen, sondern auch daraus, dass beide dafür prädestiniert sind.

4.3. Abbildung von Busphasen auf Generic Payload und TLM-Phase

Damit GP und TLM-Phase zur taktgenauen busphasenbasierten J-R-Simulation eingesetzt werden können, muss gezeigt werden, wie eine Busphase auf GP und TLM-Phase abgebildet werden kann. Diese Abbildungen nenne ich **Mapping**, insbesondere GP-Mapping und TLM-Phase-Mapping. Die folgenden Arbeitsanweisungen werden beim ersten Lesen komplex wirken. Einfache Beispiele sind wenig hilfreich, da bei diesen viele Regeln nicht wirksam werden. Daher werden die Ergebnisse der Mappings anschließend nochmals kurz zusammengefasst, da für den weiteren Verlauf der Arbeit weniger der Ablauf der Mappings, sondern vielmehr deren Ergebnis wichtig ist. In Anhang I und in Abschnitt 6.2 werden die Regeln im Detail an komplexen Protokollen erläutert, sodass der Sinn der Regeln klarer und die grundlegende Einfachheit der Abbildung deutlich wird.

4.3.1. GP-Mapping

Eine Phase ist laut Definition 3.7 mit $\Phi = (PP_{\Phi S}, PP_{\Phi E}, PP_{\Phi O}, tb_\Phi, S_\Phi, E_\Phi, A_\Phi)$ gegeben. Die Ports einer solchen Phase können mit den folgenden Schritten auf das GP abgebildet werden. Die Schritte werden nacheinander auf alle Phasen der Peripherie Φ_{Per} angewendet. Zur Vereinfachung werden GP-Grundelemente und GP-Erweiterungen als GP-Elemente bezeichnet.

Ablauf

1. Für jeden Port $p \in PP_{\Phi S} \cap PP_{\Phi E}$, der an Phasenstart und -ende beteiligt ist, führe die folgenden Schritte durch.

1.1 Überprüfe, ob die Definition aller nicht leerer Kriterien (Start-, Abbruch- oder Endkriterium) den Wert von p im Takt, in dem die Phase startet, abbricht bzw. endet, auf jeweils genau einen Wert festlegt. Ist dem so, speichere auf welchen Wert p jeweils an den Phasenrändern festgelegt wird und dass p bei Φ implizit modelliert wird und beende die Bearbeitung von Port p[1].

[1] Hintergrundinformation: Wird der Wert des Ports an den Phasenrändern auf spezielle Werte festgelegt, muss das GP diesen Wert nicht enthalten, da sich der Wert durch das Auftreten eines Phasenrandes ergibt. Wird der Wert nicht in allen Fällen festgelegt, muss er aber im GP existieren, damit er in den Fällen, in denen er nicht festgelegt ist, im GP übermittelt werden kann.

1.2 Wenn die Bitbreite von p Eins ist, überprüfe, ob die Definition des Endkriteriums den Wert von p im Takt, in dem die Phase endet, auf genau einen Wert festlegt. Ist dem so, speichere auf welchen Wert p an den Phasenrändern festgelegt wird und dass p bei Φ implizit modelliert wird und beende die Bearbeitung von Port p[2].

1.3 Suche ein GP-Element, das sinnvoll geeignet ist, die Information von Port p zu enthalten[3]. Ist eines vorhanden, vergrößere ggf. den verwendeten Datentypen in seiner Bitbreite (im Falle von GP-Erweiterungen), speichere im Falle eines GP-Grundelements, dass es zu den Phasenrändern von Φ gehört und beende die Bearbeitung von Port p.

1.4 Füge eine GP-Erweiterung zum GP hinzu. Speichere, dass die Information in dieser GP-Erweiterung im Sinne von p zu interpretieren ist. Verwende als Datentypen einen C++- oder SystemC-Datentyp, dessen Bitbreite groß genug für p ist.

2. Für jeden Port $p \in PP_{\Phi S} \backslash PP_{\Phi E}$, der einzig am Phasenstart oder -abbruch beteiligt ist, führe die Schritte 1.1 (eingeschränkt auf Start- und Abbruchkriterium), 1.3 und 1.4 durch.

3. Für jeden Port $p \in PP_{\Phi E} \backslash PP_{\Phi S}$, der einzig am Phasenende beteiligt ist, führe die Schritte 1.1 (eingeschränkt auf das Endekriterium), 1.3 und 1.4 durch.

Zusammenfassung

Nachdem obige Arbeitsschritte für alle Phasen durchgeführt wurden, enthält das GP einen Satz von Elementen, die alle eine unterschiedliche Bedeutung haben. Ports, deren Werte über An- bzw. Abwesenheit eines Phasenrandes festgelegt werden, finden sich nicht im GP wieder, sind also implizit modelliert. Für jeden anderen Port, der in einer Phase der Peripherie verwendet wurde, gibt es genau ein Element, das dessen Information widerspiegelt.

Die Schritte 1.3 und 1.4 des obigen Ablaufes enthalten einen Freiheitsgrad. Die Entscheidung, welches GP-Element sinnvoll für die Übertragung eines Portwertes geeignet ist, liegt im Ermessensspielraum des Designers. Verschiedene Designer werden unter Umständen zu unterschiedlichen Sätzen von GP-Erweiterungen gelangen, wenn sie den gleichen Satz von Busphasen auf das GP abbilden. Da dies aber für eine Kommunikationsstruktur bzw. ein MMBIF nur ein Mal geschieht und nicht wiederholt von verschiedenen Designern durchzuführen ist, ist dies vertretbar.

Es sollte bei der Abbildung stets versucht werden, die GP-Grundelemente sinnvoll mit einzubeziehen, auch wenn dies dazu führt, dass ein Port auf ein GP-Grundelement und

[2] Hintergrundinformation: Wird der Wert eines Boolschen Signals beim Ende festgelegt, ist sein Wert auch durch Abwesenheit des Endes als zum Endwert inverser Wert gegeben und muss nicht explizit im GP übermittelt werden.

[3] Ein Port, der die Adresse eines Transfers übermittelt, sollte z.B. sinnvoller Weise auf das GP-Grundelement `m_address` abgebildet werden. Zeigt ein Port z.B. an, dass die Transaktion gepuffert werden darf, sollte er auf eine GP-Erweiterung abgebildet werden, die z.B. `bufferable` heißt. Ports anderer Busphasen, die auch anzeigen, dass die Transaktion gepuffert werden kann, sollten dann auf die gleiche GP-Erweiterung abgebildet werden.

eine GP-Erweiterung abgebildet wird (anstatt nur eines von beidem, wie in Schritt 1.3). Als Beispiel hierfür sei der OCP-Command-Port genannt: OCP kennt sieben verschiedene Kommandos. Davon sind drei Lesezugriffe und vier Schreibzugriffe. Die Funktion des Ports ist also, einen dieser sieben Zugriffe zu benennen. Dies passt nicht unmittelbar mit dem `m_command`-Feld des GP zusammen (es unterscheidet nur zwischen Schreiben und Lesen). Jedoch kann ein Teil der Information auf `m_command` abgebildet werden, nämlich ob es sich um einen lesenden oder schreibenden Zugriff handelt. Eine GP-Erweiterung muss dann nur noch die Zusatzinformation, um welchen der drei lesenden bzw. der vier schreibenden Zugriffe es sich genau handelt, enthalten. Ein solches Vorgehen erhöht die Nähe zum BP, eine wichtige Voraussetzung zur Vereinfachung von Adaptern.

4.3.2. TLM-Phase-Mapping

Mit den folgenden Schritten werden die Busphasen der Peripherie auf TLM-Phasen abgebildet. Als Ziel soll ein Satz TLM-Phasen bestimmt werden, der benutzt werden kann, um Starts, Enden und Abbrüche von Busphasen anzuzeigen. Man beginnt dabei mit einem leeren Satz TLM-Phasen und führt die folgenden Arbeitsschritte für jede Phase Φ der Peripherie durch, sodass nach und nach immer mehr TLM-Phasen bestimmt werden. Dabei sollen bei der Bearbeitung einer Busphase, also zum Anzeigen eines Phasenrandes dieser Busphase, wann immer möglich schon bereits bestimmte TLM-Phasen wiederverwendet werden. Der Grund dafür ist, dass wie in Abschnitt 3.2.1 erläutert, jeder Master und jeder Slave einen eigenen Satz Busphasen hat. Der Satz der TLM-Phasen soll sich aber nicht für jeden Master und Slave unterscheiden. Daher muss festgestellt werden können, wann Busphasen verschiedener Teilnehmer die gleichen TLM-Phasen verwenden können. Um dies zu ermöglichen, müssen Busphasen miteinander verglichen werden. Dazu werden die Ergebnisse des GP-Mapping verwendet, genauer der Satz der implizit modellierten Signale einer Busphase (Ergebnis der Schritte 1.1 und 1.2) und der Satz der von einer Busphase verwendeten GP-Grundelemente (Ergebnis von Schritt 1.3).

Um festzustellen, ob eine bereits definierte TLM-Phase *ph* für die Modellierung eines Randes einer Busphase Φ in Betracht kommt, muss Φ mit der Busphase Δ verglichen werden, bei deren Bearbeitung *ph* zum Satz der TLM-Phasen hinzugefügt wurde. Aus diesem Grund wird in den folgenden Arbeitsschritten zu jeder TLM-Phase *ph* genau diese Busphase Δ gespeichert und dort definierende Busphase[4] genannt.

Ablauf

1. Versuche zur einer Phase Φ eine TLM-Phase *ph* mit Präfix `BEGIN_` im Satz der TLM-Phasen zu finden, für deren definierende Busphase Δ Folgendes gilt[5]:

[4] Da diese Busphase ursprünglich zur Definition von *ph* führte.

[5] Für die erste zu bearbeitende Busphase wird dies immer fehlschlagen, da es noch keine TLM-Phasen gibt.

1.1 Die Sätze der für Φ und Δ implizit modellierten Ports stimmen in ihren Werten überein, und die Information der implizit modellierten Ports kann sinvoll als gleich angesehen werden[6].

1.2 Die Sätze der für Φ und Δ verwendeten GP-Grundelementen sind gleich.

1.3 Gehören Φ und Δ zum gleichen Master oder Slave, müssen zeitliche Überlappungen zwischen Φ und Δ ausgeschlossen sein.

2. Wurde in Schritt 1 keine TLM-Phase gefunden, definiere die TLM-Phase `BEGIN_`Φ und lege Φ als definierende Busphase für diese TLM-Phase fest. Wurde eine TLM-Phase gefunden, wird diese zur Modellierung des Starts von Φ verwendet.
3. Ist für die Phase ein Endkriterium definiert ($E_\Phi \neq \emptyset$), finde die TLM-Phase `BEGIN_X`, mit der der Start von Φ modelliert wird (siehe Schritt 2), und suche dazu möglichst die TLM-Phase `END_X`, die das gleiche Suffix `X` hat.
4. Wurde in Schritt 3 keine TLM-Phase gefunden, definiere die TLM-Phase `END_X` und lege Φ als definierende Busphase für diese TLM-Phase fest. Wurde eine TLM-Phase gefunden, wird diese zur Modellierung des Endes von Φ verwendet[7].
5. Ist für die Phase ein Abbruchkriterium definiert ($A_\Phi \neq \emptyset$), finde die TLM-Phase `BEGIN_X`, mit der der Start von Φ modelliert (siehe Schritt 2) wird, und suche dazu die TLM-Phase `ABORT_X`, die das gleiche Suffix `X` hat.
6. Wurde in Schritt 5 keine TLM-Phase gefunden, definiere die TLM-Phase `ABORT_X` und lege Φ als definierende Busphase für diese TLM-Phase fest. Wurde eine TLM-Phase gefunden, wird diese zur Modellierung des Abbruchs von Φ verwendet[7].

Zusammenfassung

Nach dem TLM-Phase-Mapping existiert ein Satz von TLM-Phasen, mit dem die Ränder von Busphasen angezeigt werden können. Eine TLM-Phase repräsentiert alle Busphasenränder, an denen die gleichen GP-Grundelemente beteiligt sind und die in ihren impliziten Signalen (also den Werten der Ports, die keine Entsprechung im GP haben) übereinstimmen. Die Identifikation, welche Busphase (also von welchem Master oder Slave) und somit auch welche Ports der Kommunikationsstruktur genau durch ein GP und eine TLM-Phase modelliert sind, erfolgt über die Feststellung, welcher Master oder Slave das GP sendet bzw. empfängt. Es ist wichtig festzustellen, dass sich Busphasen in ihrem GP-Mapping bezüglich der verwendeten GP-Erweiterungen unterscheiden können, jedoch im TLM-Phase-Mapping auf die gleiche TLM-Phase abgebildet werden. Die TLM-Phase allein macht also keine Aussage, welche GP-Erweiterungen an der modellierten Busphase beteiligt sind. Dies begründet sich wie folgt: Wie in 4.5.2 erläutert werden wird, können sich Interfaces in Signalen, die auf GP-Erweiterungen abgebildet werden, unterscheiden und trotzdem kompatibel sein. Bezöge man

[6] Auch hier ist wieder ein Freiheitsgrad enthalten. Ob die Informationen zweier Ports als sinnvoll gleich angesehen werden können, liegt im Ermessensspielraum des Designers.

[7] Dies stellt sicher, dass Busphasen immer durch eine TLM-Phasengruppe (`BEGIN_X`, `ABORT_X`, `END_X`) mit gleichlautendem `X` repräsentiert werden.

diese Signale zur Bestimmung der TLM-Phasen heran, ergäbe sich dann ein größerer Satz an TLM-Phasen (da sich die Busphasen verschiedener Interfaces dann ggf. in diesen Signalen unterscheiden). Wie Abschnitt 4.5.1 zeigen wird, ist ein großer Satz von (fast gleichen) TLM-Phasen aber nachteilig.

Im Anschluss an das TLM-Phase-Mapping sollte geprüft werden, ob eine Phase aus dem Satz der TLM-Phasen sinnvoll als „Request“, im Sinne einer Schreib- bzw. Leseanfrage von einem Master and einen Slave und ob eine TLM-Phase sinnvoll als „Response“, im Sinne einer Antwort auf den vorhergehenden Request, bezeichnet werden kann. Ist dem so, sollten die entsprechenden TLM-Phasen `BEGIN/END/ABORT_REQ` bzw. `BEGIN/END/ABORT_RESP` genannt werden, um die im TLM-Phasentyp vordefinierten Namen zu verwenden und eine größtmögliche Nähe zum Baseprotocol zu erzielen.

4.3.3. Mehrfaches GP- und TLM-Phase-Mapping

Das oben beschriebene GP- und TLM-Phase-Mapping wird auf einen gegeben Kommunikationsautomaten und dessen Peripherie angewendet. Abhängig von der Peripherie kann es sein, dass nicht alle Aspekte des MMBIF des Kommunikationsautomaten verwendet werden. So unterscheidet sich ein PLB mit nur einem Master von einem PLB mit mehreren Mastern. Darüber hinaus kennen PLB, OCP und auch AMBA verschiedene Konfigurationen. D.h. nicht alle Signale des MMBIF müssen von Mastern und Slaves verwendet werden. In diesem Fall muss das GP- und TLM-Phase-Mapping wiederholt auf mehrere Modelle mit unterschiedlicher Peripherie angewendet werden[8], sodass alle möglichen Konfiguration abgedeckt und somit alle Aspekt des MMBIF dem GP- und TLM-Phase-Mapping zugeführt wurden.

4.3.4. TLM-Phasen-Reduzierung

Wie im TLM-Phase-Mapping erläutert, bestimmt eine TLM-Phase die Gültigkeit der GP-Grundelemente und die Werte implizit modellierter Signale. Bei konfigurierbaren MMBIF ist es möglich, dass sich Interfaces verschiedener Module der Peripherie in Ports unterscheiden, die beim GP-Mapping auf GP-Grundelemente abgebildet oder implizit modelliert werden. Dementsprechend werden dann beim TLM-Phase-Mapping verschiedene TLM-Phasen erzeugt, damit die TLM-Phasen anzeigen können, welche GP-Grundelemente gültig sind bzw. welche impliziten Werte gesetzt werden. Dies führt dann zu einem Satz sehr ähnlicher TLM-Phasen, die sich in den verwendeten GP-Elementen und impliziten Signalen unterscheiden. Wie Abschnitt 4.5.1 zeigen wird, ist aber ein großer Satz von fast gleichen Phasen nachteilig, es ist ratsam den Satz der TLM-Phasen zu reduzieren. Zwei TLM-Phasen können unter folgenden Umständen zu einer Phase zusammengelegt werden:

- Beide TLM-Phasen haben das gleiche Präfix (`BEGIN_`, `END_` oder `ABORT_`)

[8] Diese mehrfache Anwendung kann, ebenso wie das erste Mapping, für eine virtuelle Peripherie durchgeführt werden. Es ist nicht notwendig die Peripherie tatsächlich so aufzubauen.

- Beide TLM-Phasen widersprechen sich nicht in den implizit modellierten Signalen[9], es können aber unterschiedliche Signale implizit modelliert werden.

- Es gibt mindestens eine GP-Erweiterung oder ein GP-Grundelement, deren Phasenassoziation (siehe Abschnitt 4.7.1) beide TLM-Phasen enthält. Dies stellt sicher, dass die TLM-Phasen ähnliche Informationen tragen.

- Auf einem gegebenen Initiatorsocket-Targetsocket-Paar kann im Verlauf jeder möglichen, legalen Transaktion nur eine der beiden TLM-Phasen beobachtet werden. Das heißt, der Sender weiß, welche der beiden TLM-Phasen er übermitteln muss, und der Empfänger weiss schon vorher, dass es nur diese TLM-Phase sein kann. Ggf. sind zum Erlangen dieser Kenntnis besondere Mechanismen notwendig.

Als Beispiel dafür soll das OCP dienen. Das Beispiel beschränkt sich dabei auf die Signale, die implizit oder mit GP-Grundelementen modelliert werden. Im vollen OCP gibt es dazu noch mehr Signale, die auf GP-Erweiterungen abgebildet werden. Man betrachte die OCP-Requestphase mit Accept-Handshake. Es handelt sich dabei um eine Multitaktphase (siehe Anhang B). Die Signale, die der Sender treibt, sind `MCmd`, `MAddr`, `MBE`, `MData` und `MBurstSeq`. Der Empfänger treibt lediglich das Signal `SCmdAccept`. Die Phase startet, wenn `MCmd` von Null auf einen von Null verschiedenen Wert wechselt, und endet, wenn `MCmd` von Null verschieden ist und gleichzeitig `SCmdAccept` auf Ein ist.

Das Ergebnis des GP-Mappings für diese Phase zeigt Tabelle 4.2.

OCP-Signal	GP-Mapping
`MCmd`	`m_command`
`MAddr`	`m_address`
`MBE`	`m_byte_enable`
`MData`	`m_data`
`MBurstSeq`	`m_streaming_width`
`SCmdAccept`	implizit modelliert

Tabelle 4.2.: GP-Mappingergebnis für die OCP-Requestphase

Das TLM-Phase-Mapping wird danach dazu führen, dass die TLM-Phasen `BEGIN_REQ` und `END_REQ` verwendet werden. Erstere markiert die in Tabelle 4.2 gezeigten GP-Grundelemente als gültig. Letztere modelliert implizit, dass `SCmdAccept` auf logisch Eins gesetzt wird.

Nun handelt sich beim OCP aber um ein konfigurierbares Interface, und alle Signale der Request-Phase außer `MCmd` und `SCmdAccept` können unabhängig voneinander aus dem Interface entfernt werden. Dies ist abhängig davon, welche Art von OCP-Kommunikation

[9] D.h. modelliert die eine Phase einen Port implizit und besitzt die andere Phase einen Port mit sinnvoll gleicher Bedeutung, dann modelliert sie diesen Port auch implizit mit dem gleichen Wert.

erwünscht ist. Dadurch ergeben sich 16 verschiedenen Varianten für Phasenstartports der Phase[10]. Beim TLM-Phase-Mapping werden dann 32 verschiedene TLM-Phasen entstehen, wobei eine davon die oben erläuterte Phase `BEGIN_REQ` und eine andere `END_REQ` ist. 15 weitere TLM-Phasen entsprechen `BEGIN_REQ`, markieren aber andere Kombinationen von GP-Elementen als gültig, und die restlichen 15 TLM-Phasen sind die entsprechenden `END_`-Phasen

Jedoch erfüllen alle 16 `BEGIN_REQ`-Varianten die oben erwähnten Anforderungen und können zu einer einzigen `BEGIN_REQ`-Phase und somit auch `END_REQ`-Phase zusammengelegt werden. Damit auf einem Initiatorsocket-Targetsocket-Paar im Vorhinein bekannt ist, welche GP-Elemente durch `BEGIN_REQ` als gültig markiert werden, also welche Variante von `BEGIN_REQ` auf diesem Socketpaar zulässig ist, müssen die verbundenen Sockets Informationen darüber austauschen, welche OCP-Signale Teil ihrer OCP-Konfiguration sind (siehe Abschnitt 4.6.4).

4.4. Eignung und Regeln zur Verwendung von `nb_transport`

Unter Verwendung von GP und TLM-Phase verändert sich die Signatur des in Abschnitt 4.2 eingeführten IMC `announce` wie in Listing 4.3 in den Zeilen 1 und 2 dargestellt. Abschnitt 3.3 hat verdeutlicht, dass die Komplexität von Adaptern für die Mixed-Mode-Simulation geringer wird, je näher der taktgenaue Modellierungsstandard an die abstrakteren Standards angelehnt ist. Es sollte also vermieden werden, einen völlig neuen IMC zu definieren. Idealerweise wird möglichst ein bereits existierender wiederverwendet.

Vergleicht man die Signatur von `announce` mit den bereits in TLM-2.0 existierenden IMC `b_transport` und `nb_transport` unter Verwendung von GP und TLM-Phase, erkennt man, dass `nb_transport` der Signatur von `announce` sehr nahe (Listing 4.3, Zeilen 4 und 5) kommt, während `b_transport` aufgrund der fehlenden TLM-Phase nicht in Frage kommt. Jedoch kennt `nb_transport` noch ein drittes Argument, einen Rückgabewert und eine Vielzahl bereits in TLM-2.0 definierter Regeln. Zur endgültigen Entscheidung über die Eignung von `nb_transport` werden diese im Folgenden untersucht.

```
1 void
2 announce    (tlm_generic_payload&, tlm_phase&};
3
4 tlm_sync_enum
5 nb_transport(tlm_generic_payload&, tlm_phase&, sc_time&};
```

Listing 4.3: Gegenüberstellung: `announce-nb_transport`

[10]Die Anwesenheit eines konfigurierbaren Signales im Interface kann als binäre Variable angesehen werden. Vier konfigurierbare Signalen ergeben folglich $2^4 = 16$ Möglichkeiten.

Die vorangegangenen Abschnitte haben gezeigt, dass man `nb_transport` zusammen mit dem GP und der TLM-Phase grundsätzlich zur Übergabe der Inputs und Ouputs an bzw. von einem taktgenau *J-R*-simulierten Kommunikationsautomaten verwenden kann.

Die von mir definierte, taktgenaue busphasenbasierte *J-R*-Simulation ist weniger abstrakt als der im TLM-2.0-Standard definierte Modellierungsstil Approximately Timed (AT), welcher `nb_transport` verwendet. Benutzte man `nb_transport` auch für die taktgenaue Modellierung, stimmten die Interfaces für AT- und taktgenaue busphasenbasierte Modellierung überein.

Es ist nicht im Sinne des Anwenders, bei Verwendung des gleichen Interfaces[11] dessen Verwendungsregeln grundlegend umzudeuten. Ist dies nötig, sollte ein neues Interface definiert werden, um diese grundlegenden Unterschiede zu verdeutlichen[12].

Im folgenden Abschnitt werden die Regeln aus dem TLM-2.0-Language-Reference-Manual (LRM) bezüglich von `nb_transport` aufgelistet und deren Anwendbarkeit für taktgenaues busphasenbasiertes TLM diskutiert. Danach kann entschieden werden, ob `nb_transport` für taktgenaues TLM eingesetzt werden sollte, oder ob ein neuer IMC erforderlich ist.

Die Listung und Diskussion der Regeln wird stets den entsprechenden Abschnitt des TLM-2.0-LRM $[\text{OSCI09a}]_i$ zitieren. Absätze von LRM-Abschnitten, die Nebeninformationen enthalten oder Korollare ziehen, werden nur diskutiert, wenn sich die zu Grunde liegende Regel im Kontext der taktgenauen Modellierung geändert hat.

Die protokollunabhängigen Regeln zum Non-blocking-Transport-Interface (NBTI) sind in Abschnitt 4.1.2 und 4.1.3 von $[\text{OSCI09a}]_i$ hinterlegt. Dabei bedeutet protokollunabhängig, dass die Typen der Argumente `trans` und `phase` (vgl. Abschnitt 2.3.2), noch nicht festgelegt sind. Die Festlegung dieser Typen erfolgt erst durch ein spezielles Protokoll (z.B. dem Base-Protocol). Die Regeln untergliedern sich in Aufrufkonventionen, Regeln zu den einzelnen Argumenten und dem Rückgabewert.

Aufrufkonventionen

Die Konventionen zum Aufruf von `nb_transport` sind in Abschnitt 4.1.2.4 des TLM-2.0-LRM enthalten. Darin werden unter Anderem die Aufrufrichtungen (`nb_transport_fw` nur vom Initiator-Socket zum Target-Socket, `nb_transport_bw` entgegengesetzt), die nicht-blockende Eigenschaft (kein `sc_core::sc_wait` innerhalb von `nb_transport`) und das Verbot `nb_transport_fw` und `nb_transport_bw` ineinander verschachtelt aufzurufen, festgelegt.

Diese Konventionen müssen bei der taktgenauen Modellierung nicht geändert werden. Jedoch werden zusätzliche Konventionen notwendig, welche in Abschnitt 4.9 erläutert werden.

[11] im C++-Sinne stimmen beiden Schnittstellen in Anzahl und den Namen und Signaturen der in ihnen enthaltenen Funktionen überein

[12] im C++-Sinne also die Übereinstimmung in Signaturen erhalten, die Übereinstimmung im Namen aber auflösen

Regeln zum trans-Argument

Abschnitt 4.1.2.5 des TLM-2.0-LRM legt fest, dass die Lebensdauer des `trans`-Arguments die Dauer des Aufrufs übersteigen und so das gleiche Objekt mehrfach mittels `nb_transport` übertragen werden kann. Darauf aufbauend kann dann eine Transaktion als Sequenz von `nb_transport` Aufrufen mit dem gleichen `trans`-Argument modelliert werden. Darüber hinaus wird festgelegt, dass sowohl Aufrufer als auch Besitzer von `nb_transport` den Inhalt von `trans` modifizieren können und geeignete Regeln für einen speziellen `trans`-Typ existieren müssen, die diese Zugriffe koordinieren.

All diese Regeln müssen bei der taktgenauen Modellierung nicht geändert werden.

Regeln zum phase-Argument

Die Regeln in Abschnitt 4.1.2.6 des TLM-2.0-LRM legen fest, dass das `phase`-Argument einen Schritt der Kommunikation bestimmt. Das Argument soll genutzt werden, um die aktuell zulässigen Zugriffe auf das `trans`-Argument zu steuern und den Zustand der Kommunikation[13] auf Seite des Aufrufers anzuzeigen. Da eine Transaktion auf dem Weg vom Initiator zum Target mehrere Initiator/Target-Socket-Paare durchlaufen kann, kann die Kommunikation auf all diesen Paaren potentiell in verschiedenen Zuständen sein. Dementsprechend wird festgelegt, dass das `phase`-Argument im Gegensatz zum `trans`-Argument nicht vom Initiator bis zum Target übermittel wird, sondern dass jeder Aufrufer von `nb_transport` ein eigenes `phase`-Objekt besitzt und dass die Lebensdauer dieses Arguments nur der Dauer des Aufrufs entspricht. Auch diese Regeln müssen bei der taktgenauen Modellierung nicht geändert werden.

Regeln zum Rückgabewert

Die Regeln zum Rückgabewert, der vom Typ `tlm_sync_enum` ist, werden in Abschnitt 4.1.2.7 des TLM-2.0-LRM beschrieben und wurden bereits in Abschnitt 2.3.2 in der Beschreibung des Rückgabewertes zusammengefasst.

Der Rückgabewert `TLM_ACCEPTED` ist unproblematisch. `TLM_UPDATED` ist prinzipiell auch in der taktgenauen Modelierung von Wert[14]. Als problematisch erachte ich `TLM_COMPLETED`. Mit diesem Rückgabewert signalisiert der Besitzer von `nb_transport` dem Aufrufer, dass die Kommunikation abgeschlossen ist. Dies kann prinzipiell bei jedem beliebigen Aufruf von `nb_transport` erfolgen, darf aber vom modellierten Protokoll eingeschränkt werden. Z.B. darf im BP auf den Aufruf von `nb_transport_bw` mit der Phase `END_REQ` nicht mit `TLM_COMPLETED` geantwortet werden (siehe Abbildung 2.11).

An welchen Stellen kann bei taktgenauer Modellierung sinnvoll mit `TLM_COMPLETED` geantwortet werden? Bei allen von mir untersuchten Protokollen ist dies nur bei der letzten Phase einer Transaktion möglich. Vorher haben die Kommunikationspartner entweder noch nicht

[13] in der Regel ist dies der Zustand des modellierten Protokoll-Zustandsautomaten
[14] siehe Abschnitt 4.9

ausreichend Daten ausgetauscht, um die Transaktion zu beenden oder, wenn die Daten ausreichen, kann eine Rückgabe von `TLM_COMPLETED` die Aufrechterhaltung der Taktgenauigkeit zerstören. Um dies zu verdeutlichen, zeigt Abbildung 4.4 einen OCP-Single-Request-Multiple-Data-Read-Burst in einer vereinfachten RTL-Signaldarstellung und mit den entsprechenden, vereinfachten `nb_transport`-Aufrufen. Unter der Annahme, dass der Slave alle drei angeforderten Worte auf einmal in das `trans`-Argument eintragen könnte und diese auch alle zum gleichen Zeitpunkt parat hat, könnte er auf `BEGIN_REQ` mit `TLM_COMPLETED` antworten. Jedoch kann weder der Slave noch der Master den Endzeitpunkt der Transaktion berechnen. Dem Master fehlen dazu die exakten Zeitpunkte von `BEGIN_RESP`, dem Slave fehlen dazu die noch nicht kommunizierten Zeitpunkte von `END_RESP`. Auch kann der Slave die Zeitpunkte von `BEGIN_RESP` nicht in das `trans`-Argument eintragen, da er zu deren Bestimmung die Zeitpunkte von `END_RESP` benötigt. Erst beim letzen `BEGIN_RESP` könnte der Slave und beim letzten `END_RESP` der Master mit `TLM_COMPLETED` antworten.

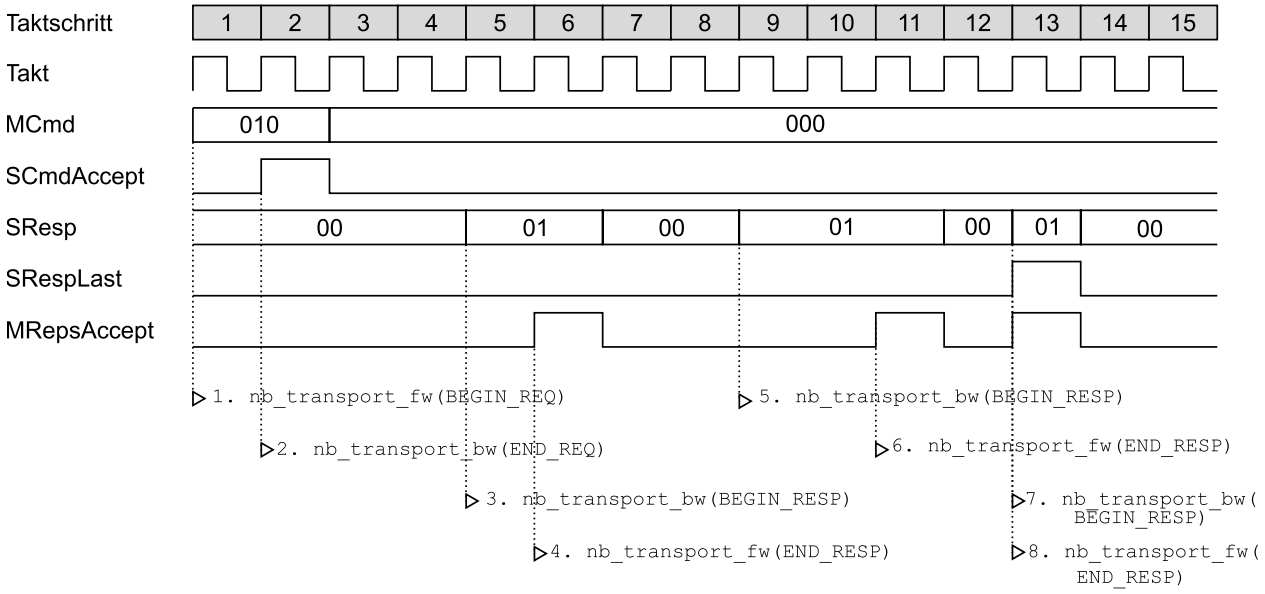


Abbildung 4.4.: Beispiele für `nb_transport`-Aufrufe in einem OCP-Read-Burst

Auch bei Protokollen, bei denen die Burstprogressionskontrolle einseitig realisiert und so eine Eintragung der Pro-Wort-Verzögerungen in das `trans`-Argument möglich wäre, führte `TLM_COMPLETED` zu einer Verkomplizierung des Modells. `TLM_COMPLETED` ist ein Sonderfall; im Normalfall müssen die Kommunikationspartner davon ausgehen, dass es nicht eingesetzt wird. Folglich existiert in jedem Kommunikationspartner eine Implementierung, bei der die Kommunikation in Takten voranschreitet. In der Regel werden dazu Zustandsautomaten verwendet, die mit der Kommunikation voranschreiten, ggf. auch synchronisiert mit weiteren Zustandsautomaten. Empfängt nun ein Kommunikationspartner ein `TLM_COMPLETED`, so bedeutet das, dass eine beliebige Anzahl von zukünftigen Takten so abgespielt werden muss, als würde kommuniziert werden. Dies erfordert zusätzlichen Code und erhöht die Fehleranfälligkeit des Modells. Darüber hinaus würde das Einfügen von Zeitinformation in das `trans`-Argument Punkt 4.1.3.1a des TLM-2.0-LRM zuwider laufen; darin wird ein solches

Vorgehen nicht empfohlen. Aus diesem Grund soll der Rückgabewert TLM_COMPLETED im Rahmen der taktgenauen Modellierung nicht verwendet werden.

Regeln zum time-Argument

Die Verwendung des time-Arguments wird in Abschnitt 4.1.3 des TLM-2.0-LRM geregelt. Prinzipiell können diese Regeln unverändert für die taktgenaue Modellierung angewendet werden, jedoch muss der Sinn der Verwendung des Arguments in diesem Fall kritisch hinterfragt werden.

Wird bei einem Aufruf von nb_transport eine von SC_ZERO_TIME verschiedene Zeit t als Argument übergeben, so bedeutet dies, dass der mit dem Aufruf modellierte Phasenrand[15] nicht zum aktuell simulierten Zeitpunkt now=sc_time_stamp(), sondern erst zum Zeitpunkt now+t als eingetreten angesehen werden kann. Man kann also davon sprechen, dass der Phasenrand in die simulierte Zukunft projiziert wird.

Im Rahmen der taktgenauen Modellierung ist die Zeitbasis für eine Kommunikationsstruktur ein Takt. Dementsprechend drückt ein Kommunikationsteilnehmer, sei dies ein Element der Peripherie oder der Kommunikationsautomat, eine Verzögerung in der Regel als eine Anzahl Takte aus. Um nun das time-Argument zu verwenden, muss der Sender diese Information in absolute Zeit konvertieren (Multiplikation der Takte mit der Taktperiode) und der Empfänger dies wieder rückgängig machen (Division des time-Arguments durch die Taktperiode). Dies ist eine unerwünschte Zusatzlast für den simulierenden Rechner.

Im Rahmen der taktgenauen Modellierung ist eine Zukunftsprognose bezüglich der Kommunikation nur schwer zu treffen. Existiert nur eine Schnittstelle an einem IP-Block, deren Verhalten vom IP-Block selbst nicht prognostizierbar oder kontrollierbar ist und die Einfluss auf die MMBIF-Kommunikation haben kann, wird das Projizieren von Kommunikation in die Zukunft ohne Zusatzmechanismen unmöglich. Bei vielen IP-Blöcken auf taktgenauer Ebene ist eine solche Schnittstelle durch die Existenz eines System-Resets gegeben, sodass diese IP-Blöcke das time-Argument nicht ohne weiteres nutzen können. Um das zu ermöglichen, müsste eine Möglichkeit geschaffen werden, nb_transport-Aufrufe zu widerrufen. Dies könnte über einen anti_transport-IMC oder über zusätzliche TLM-Phasen (z.B. UNDO_BEGIN_REQ) realisiert werden. In jedem Fall führt dies zu einer Komplexitätssteigerung im Interface und so auch in der Implementierung von Modellen, die das Interface verwenden.

Um also eine derartige Komplexitätssteigerung und auch die oben erwähnten unerwünschten Umrechnungen zu vermeiden, empfehle ich das time-Argument im Rahmen der taktgenauen Modellierung grundsätzlich nicht zu verwenden. Einzig wenn ein mittels des time-Arguments in die Zukunft projizierter nb_transport-Aufruf unter keinen Umständen widerrufen werden muss, kann eine Verwendung des Arguments in Betracht gezogen werden.

[15] z.B. Beginn, Ende oder Abbruch einer Phase

Zusammenfassung

Die protokollunabhängigen Regeln für das NBTI werden bis auf die Zulässigkeit des Rückgabewertes `TLM_COMPLETED` auch für die taktgenaue Modellierung übernommen. Für die Zeitannotation wird die Empfehlung ausgesprochen, diese nicht einzusetzen. Die Regeln werden also lediglich in einem Punkt eingeschränkt, eine Verwendung von `nb_transport` zur taktgenauen Modellierung ist somit ohne grundlegende Umdeutungen möglich, es muss also kein neuer IMC für die taktgenaue Modellierung eingeführt werden.

4.5. Regeln für Generic Payload und TLM-Phase

Als Folgerung aus den Abschnitten 4.2 und 4.4 ergibt sich, dass zur taktgenauen busphasenbasierten Modellierung `nb_transport` mit GP und TLM-Phase eingesetzt werden kann. Während die protkollunabhängigen Regeln für `nb_transport` bis auf eine Einschränkung übernommen werden können, ist noch nicht geklärt, welche Regeln für den Zugriff auf das GP und die TLM-Phase im Rahmen der taktgenauen Modellierung zum Tragen kommen. Idealerweise sollten diese nahe an den Regelungen zum TLM-2.0-BP liegen, sodass ein Erlernen der taktgenauen Modellierung bei bereits vorhandenen Kenntnissen über das BP vereinfacht wird. Jedoch sind die Regelungen zum BP ganz klar für die LT- und AT-Modellierung ausgelegt. Im Folgenden wird gezeigt, dass neue, protokollabhängige Regeln notwendig sind.

4.5.1. Regeln zur TLM-Phase

Im Rahmen der taktgenauen Modellierung werden in der Regel mehr Phasen notwendig als im BP (z.B. eine Datenphase). Das Hinzufügen von Phasen ist in TLM-2.0 bereits vorgesehen und kann auf die gleiche Art und Weise vorgenommen werden, wie in Punkt 8.2.5a des TLM-2.0-LRM („Hinzufügen nicht-ignorierbarer Phasen“) beschrieben. Somit ist die Verwendung der TLM-Phase, auch im Falle von neuen Phasen, prinzipiell genügend im TLM-2.0-LRM beschrieben. Jedoch gibt es im Rahmen der taktgenauen Modellierung eine Besonderheit, die im BP ausdrücklich verboten und daher auch dort nicht weiter betrachtet wird: TLM-Phasen können sich im Laufe einer Transaktion mehrfach wiederholen. Während bei der AT-Modellierung eine TLM-Phase nur einmal pro Transaktion auftreten darf und somit den Beginn oder das Ende einer ganze Gruppe von Busphasen repräsentiert, kann diese bei der taktgenauen Modellierung in mehrere Wiederholungen der TLM-Phase, nämlich in Starts, Enden oder Abbrüche der einzelnen Busphasen, zerfallen. In Abbildung 4.5 wird dies mit Hilfe eines OCP-Multiple-Request-Multiple-Data-Read-Burst illustriert.

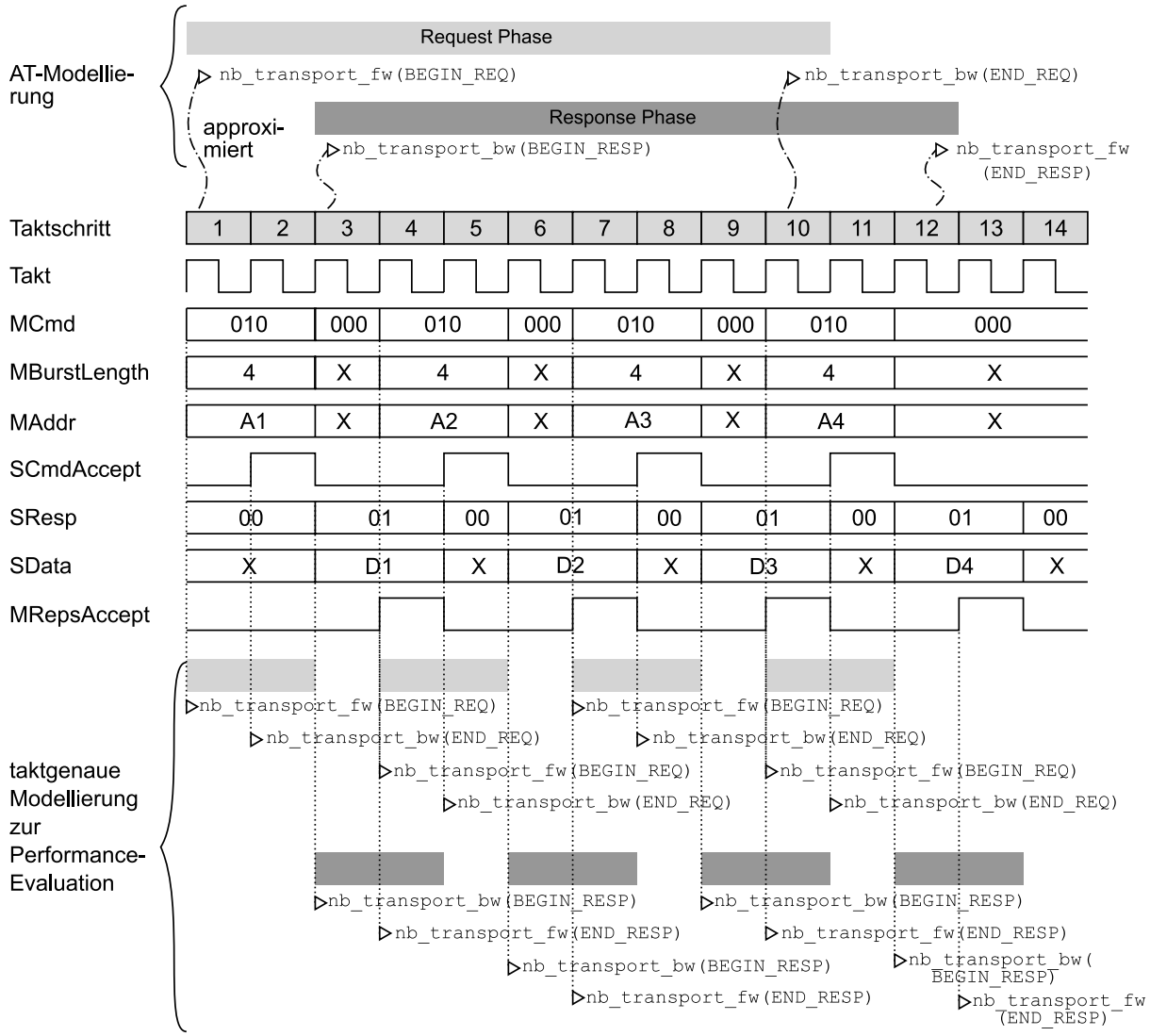


Abbildung 4.5.: Ein OCP-MRMD-Read-Burst in RTL, AT- und taktgenauer Modellierung

Lässt man eine derartige Phasenwiederholung zu, hat dies Auswirkungen auf die Regelungen zum Generic Payload. Wie in Abschnitt 4.4, Absatz „Regeln zum `phase`-Argument“, erwähnt, steuert die Phase, welche Elemente des GPs gültig sind, gelesen oder geschrieben werden dürfen. Darf eine Phase mehrfach während einer Transaktion auftreten, ist die Phase allein für diese Steuerung nicht mehr ausreichend. Die Information, um die wievielte Wiederholung der Phase es sich handelt, ist dann auch von Bedeutung (Vorgriff auf Abschnitt 4.5.2), die Zugriffregeln auf das GP werden komplexer. Kann also die Wiederholung von Phasen vermieden werden, um die Zugriffregeln möglichst einfach und nahe an der AT-Modellierung zu halten?

Alternative: Nummerierte TLM-Phasen

Die einzige Alternative, die Wiederholungen einer TLM-Phase zu vermeiden, ist für jede Wiederholung eine eigene, eindeutige TLM-Phase zu definieren. Abbildung 4.6 zeigt diesen Ansatz unter Wiederverwendung des Beispiels aus Abbildung 4.5.

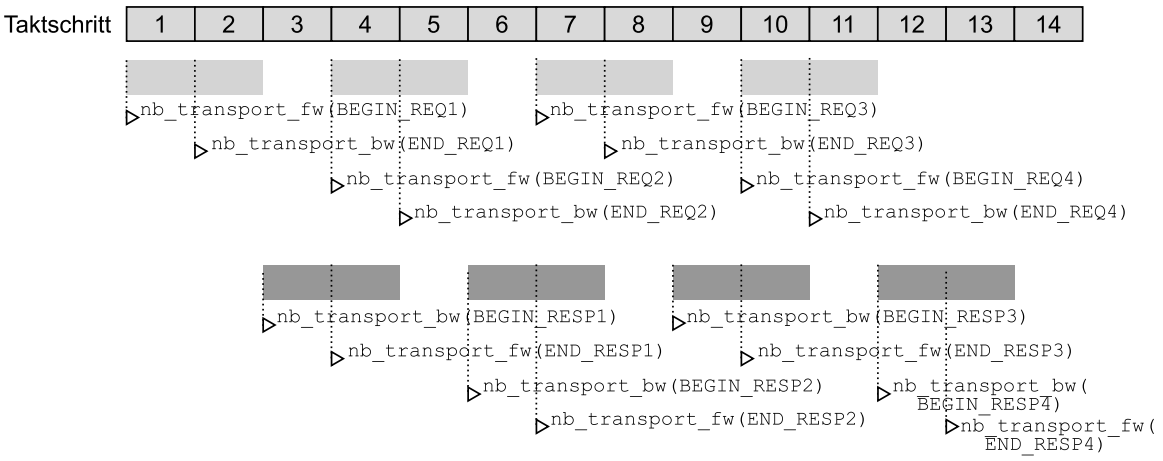


Abbildung 4.6.: Phasenwiederholung mit nummerierten TLM-Phasen

Mit diesem Ansatz können die Zugriffsregeln auf das GP wieder einzig von der TLM-Phase abhängig gemacht werden, da das Zählen der Wiederholungen in der TLM-Phase selbst codiert ist. Da die einzelnen TLM-Phasen per TLM-2.0-LRM Singleton-Objekte (siehe [GHJV94]) sind, müssen sie vor dem Kompilieren der Simulation definiert werden. Demzufolge kann nur eine endliche, zur Kompilierzeit festgelegte Anzahl von Phasen existieren. Es gibt aber MMB-Proktolle, wie z.B. Processor-Local-Bus (PLB), AHB und OCP, bei denen eine Transaktion prinzipiell beliebig lang sein und so aus beliebig vielen Phasen bestehen kann. Die vorgestellte Alternative würde bei diesen Protokollen die Modellierung solcher Transaktionen auf die einschränken, bei denen die Anzahl der Phasen die vordefinierte Anzahl nicht übersteigt.

Darüber hinaus ist der einzig mögliche Vergleich mit der TLM-Phase der Test auf Gleichheit. Um eine empfangene TLM-Phase auf Zugehörigkeit zu einer spezielle Gruppe von TLM-Phasen zu testen, muss die TLM-Phase mit allen TLM-Phasen der Gruppe einzeln verglichen werden. Listing 4.7 verdeutlicht dies. Dort wird getestet, ob die empfangene TLM-Phase eine `BEGIN_RESP` mit gerader Zählernummer ist. Solch ein Test ist z.B. notwendig, wenn ein Busbreitenkonverter implementiert wird, der einen 64-Bit-Bus auf einen 32-Bit-Bus übersetzt. In diesem Fall muss der Konverter nur jedes zweite, also gerade, `BEGIN_RESP` vom 32-Bit-Bus zum 64-Bit-Bus weiterleiten.

```
1 //TLM-Phase ph wurde empfangen
2 if (ph==BEGIN_RESP2 || ph==BEGIN_RESP4 || ph==BEGIN_RESP6 || ... || ph==BEGIN_RESP100)
3   std::cout<<"Eine gerade BEGIN_RESP-Phase wurde empfangen"<<std::endl;
4 else
5   std::cout<<"Eine ungerade BEGIN_RESP-Phase wurde empfangen"<<std::endl;
```

Listing 4.7: Test auf eine „gerade“ `BEGIN_RESP`-Phase bei nummerierten TLM-Phasen

Man erkennt in Listing 4.7, dass es im Beispiel maximal 100 `BEGIN_RESP`-Phasen geben kann. Die `if`-Anweisung muss also bis zu 50 Vergleiche durchführen. Die Anzahl der notwendigen Vergleiche hängt von der maximal möglichen Anzahl von Phasen in einer Transaktion ab. Reduziert man diese, werden die Vergleiche effizienter, die oben beschriebene Einschränkung der modellierbaren Transaktionen aber extremer. Andersherum: Erlaubt man die Mo-

dellierung langer Transaktionen, werden die Vergleiche ineffizient. Ein ähnliche Szenarios ergibt sich auch beim phasenabhängigen Zugriff auf das GP. In diesem Fall müssten 100 kaskadierte `if-else`-Anweisungen eingesetzt werden, um festzustellen, welche Phase empfangen wurde.

Aus diesen Gründen kann die nummerierte TLM-Phase nicht sinnvoll eingesetzt werden.

Fazit

Die einzig mögliche Alternative zur Vermeidung von TLM-Phasenwiederholungen kann nicht eingesetzt werden. Die Regel im TLM-2.0-LRM bezüglich der TLM-Phase müssen also derart geändert werden, dass TLM-Phasen wiederholt werden dürfen, und die Sender und Empfänger von TLM-Phasen müssen diese geeignet zählen. Prinzipiell ist es möglich, die Signatur von `nb_transport` um einen Phasenzähler zu erweitern. Davon wird aus folgendem Grund abgesehen: Ein solcher Zähler kann auch als Punkt-zu-Punkt-variable GP-Erweiterung (Vorgriff auf Abschnitt 4.7) umgesetzt werden und erlaubt so die direkte Verwendung des bereits im TLM-2.0-Standard vorhandenen `nb_transport`-IMC.

4.5.2. Regeln zum Generic Payload

Abschnitt 7 des TLM-2.0-LRM enthält Regeln zum Memory-Management des GPs und zu den Modifiabilties (Begriffserklärung 4.8), der Bedeutung und der Verwendung der GP-Grundelemente und zur Interoperabilität von Modulen, die das GP verwenden. Das Memory-Management des GP und die Bedeutung der GP-Grundelemente werden im Rahmen der taktgenauen Modellierung nicht verändert. Die Regelungen zur Interoperabilität bei Verwendung des GP und auch die Definition der Modifiabilties der GP-Grundelemente sind aber für die taktgenaue Modellierung nicht ausreichend. Die entsprechenden Regeln und Unzulänglichkeiten werden in diesem Abschnitt diskutiert.

Begriffserklärung 4.8 :
Die **Modifiabilty** (Abschnitt 7.7 des TLM-2.0-LRM; zu Deutsch „Veränderbarkeit“) eines GP-Grundelements bzw. einer GP-Erweiterung ist durch zwei Eigenschaften bestimmt: Einerseits zu welchen Zeitpunkten oder innerhalb welcher Zeitfenster ein Element geändert werden darf und andererseits ab welchem Zeitpunkt oder innerhalb welchem Zeitfenster ein Element gelesen werden darf. Änderungen eines Wertes können z.B. nur vor dem ersten `nb_transport`-Aufruf zulässig sein, während der Wert nur nach dem ersten Empfang von `nb_transport` mit einer speziell TLM-Phase gelesen werden darf.

Interoperabilität bei Verwendung des GPs durch Types-Classes

Wie in Abschnitt 2.3.2 und Anhang A beschrieben, können zwei TLM-2.0-Sockets nur verbunden werden, wenn sie die gleiche Types-Class (TC) verwenden. Nur dann kann das Modell kompiliert werden. Man spricht in diesem Zusammenhang von Bindungs-Checks zur Komiplierzeit. Die TC legt mit Hilfe von Typdefinitionen den verwendeten Transaktionstyp und

den verwendeten Phasentyp fest. Darüber hinaus bestimmt sie laut TLM-2.0-LRM Abschnitt 7.2 implizit, d.h. mit Hilfe in der Dokumentation der TC festgehaltener Regeln, welche GP-Erweiterungen das GP enthalten darf oder muss. Im Falle der `tlm_base_protocol_types`-TC sind dies ausschließlich sog. ignorierbare Erweiterungen. Das bedeutet, jedes Modul außer dem, das die Erweiterung hinzugefügt hat, darf sich so verhalten, als sei die Erweiterung nicht da.

Offenbar können mit Hilfe solcher ignorierbaren Erweiterungen keine essentiellen Protokolleigenschaften modelliert werde. Fügt ein Initiator z.B. eine GP-Erweiterung zur Signalisierung einer Priorität zum GP hinzu, so erwartet er sinnvollerweise einen gewissen Effekt. Ein Interconnect oder Target muss GPs mit Prioritäten anders behandeln als ohne. Die Erweiterung ist also nicht ignorierbar. Braucht ein Protokoll solche nicht ignorierbare GP-Erweiterungen (dies ist bei der taktgenauen Modellierung häufig der Fall), so benötigt es konsequenterweise eine neue TC, die dann festlegt, welche Erweiterungen genutzt werden.

Auf den abstrakten Ebenen ist dieser Mechanismus gut geeignet, da in der Regel nur wenige Erweiterungen benötigt werden und diese, wenn überhaupt, in jedem Modul bearbeitet werden können bzw. müssen. Bei der taktgenauen Modellierung jedoch ist dies insbesondere bei konfigurierbaren MMBIF problematisch. Solche MMBIF enthalten, für Master und Slaves verschieden, optionale Signale und entsprechende Regeln, ob und wie Interfaces verbunden werden können, wenn ein Signal in einem Interface vorhanden ist, im anderen aber nicht. Abbildung 4.9 verdeutlicht dies anhand des OCP und dessen MBurstPrecise-Signales:

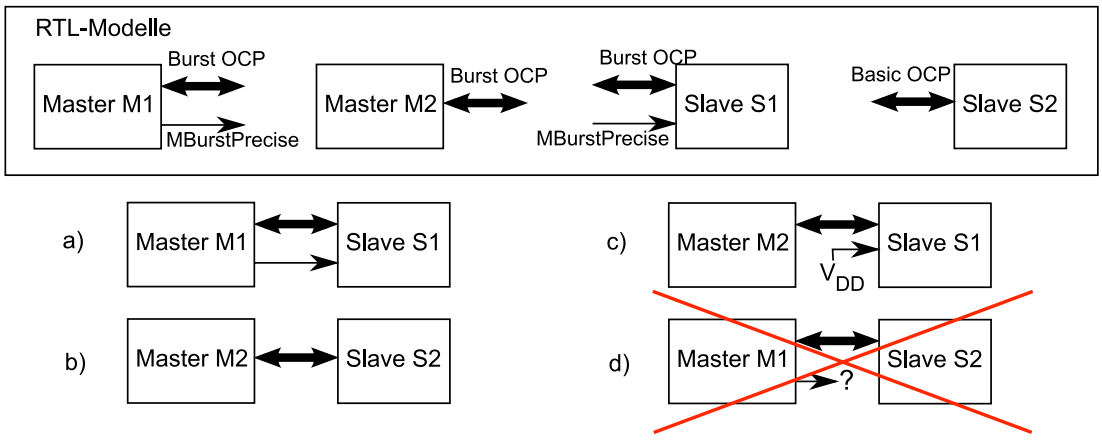


Abbildung 4.9.: Mögliche Verbindungen von OCP-Modulen, deren Interfaces sich im MBurstPrecise-Signal unterscheiden

Alle Module in Abbildung 4.9 unterstützen eine bestimmte Art von Burst-Kommunikation über OCP, in Abbildung 4.9 vereinfachend nur „Burst OCP“ genannt. Zusätzlich dazu kann der Master M1 auch noch Bursts verwenden, bei denen er zu Beginn die Länge noch nicht kennt. Dazu setzt er entweder das Signal MBurstPrecise auf Eins (d.h. die Burst-Länge ist bekannt) oder auf Null (d.h. die Burst-Länge ist nicht bekannt). Abhängig davon muss ein Slave das Burst-Längen-Feld im OCP anders interpretieren. Slave S1 kann sowohl mit Burst bekannter und unbekannter Länge umgehen, während Slave S2 ausschließlich Bursts mit bekannter Länge verarbeiten kann und Master M2 ausschließlich Bursts mit bekannter

Länge versendet. Für die Module M2 und S2 erlaubt das OCP, das Signal MBurstPrecise nicht im Interface zu führen, da es bei Abwesenheit als logische Eins angesehen wird. Somit ergeben sich die per OCP erlaubten Verbindungen wie in Abbildung 4.9a) bis d) gezeigt.

Die trivialen Verbindungen, wenn nämlich die Interfaces von Master und Slave identisch sind, sind in Abbildung 4.9a) und b) gezeigt. Abbildung 4.9c) zeigt, wie Master M2 andSlave S1 angeschlossen wird. Der MBurstPrecise-Eingang von S1 wird auf logisch Eins festgelegt, dadurch interpretiert S1 jeden Burst als Burst mit bekannter Länge. Dies entspricht genau dem, was Master M2 zu senden beabsichtigt.

In Abbildung 4.9d) wird deutlich, dass Master M1 nicht mit Slave S2 verbunden werden kann, da Slave 2 keinen MBurstPrecise-Eingang hat und so jeden Burst als Burst mit bekannter Länge interpretiert, obwohl Master M1 durchaus auch Bursts unbekannter Länge absetzen kann.

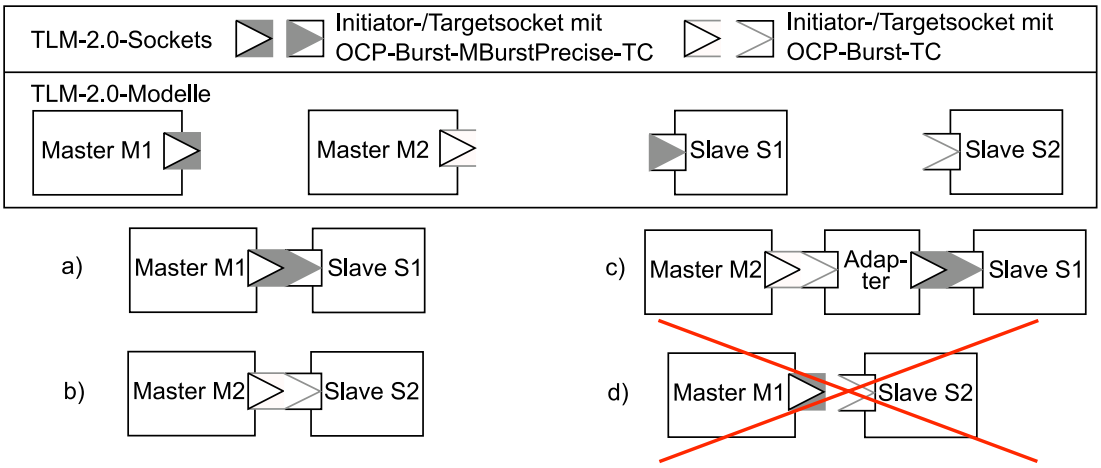


Abbildung 4.10.: Mögliche Verbindungen von OCP-Modulen verschiedener Traits-Classes

Im TLM-Phase-Mapping wird das MBurstPrecise-Signal auf eine GP-Erweiterung abgebildet werden, welche ganz klar nicht ignorierbar ist. Das bedeutet, eine neue TC ist zu definieren, welche bestimmt, dass das „Burst OCP“ und die MBurstPrecise zugeordnete GP-Erweiterung verwendet werden. Konsequenter Weise werden M1 und S1 diese TC verwenden, während M2 und S2 eine TC nutzen, die lediglich die Nutzung des „Burst OCP“ anzeigt (Abbildung 4.10). Dies ist im Sinne des TLM-2.0-LRM, birgt jedoch Probleme:

Mit oben skizzierter Verwendung der TC können die TLM-2.0-Modelle von M1 und S1 und die Modelle von M2 und S2 korrekt verbunden werden. Die nicht zulässige Verbindung von M1 und S2 wird durch die TC ebenfalls verhindert, jedoch wird auch die zulässige Verbindung von M2 und S1 ausgeschlossen. Jede andere Verteilung von TC auf die Sockets der Modelle wird stets mindestens eine gültige Verbindung verbieten oder die unzulässige erlauben. Das Erlauben der unzulässigen Verbindung ist unter keinen Umständen wünschenswert, es muss also eine gültige Verbindung (wie oben beschrieben) verboten werden. Um dann die mittels TC verbotene, gültige Verbindung wieder zu ermöglichen, ist ein Adapter notwendig, der lediglich die Verbindung wie in Abbildung 4.10c) dargestellt erlaubt. Er leitet eingehende Aufrufe einfach zur anderen Seite weiter und hat darüber hinaus keine weitere Funktion.

Solche Adapter haben Nachteile: sie verkomplizieren die Systemstruktur und führen zu Simulationsoverhead. Diese Nachteile wären bei geringer Anzahl von Adaptern akzeptierbar. Im OCP gibt es jedoch circa 16 Signale, die auf GP-Erweiterungen abgebildet werden, die unabhängig voneinander aus dem Interface entfernt werden können. Man benötigt dann im schlimmsten Fall 2^{16} verschiedene TC und mindestens einen Adapter pro solcher TC. In der Realität sind es meist mehr Adapter pro TC, da eine TC zu vielen anderen kompatibel ist.

Damit ist die Wahrscheinlichkeit, Adapter im System zu benötigen, sehr hoch, und die Nachteile der Adapter werden bemerkbar. Außerdem ist der Wartungsaufwand für eine derart große Anzahl von Adaptern, selbst bei trivialem Aufbau, nicht von der Hand zu weisen.

Weitere Probleme bei derartiger Verwendung von TC ergeben sich für sog. generische IP-Modelle. Solche IP-Modelle sind in ihrem Interface und somit in ihrem Verhalten nicht absolut festgelegt. Im Rahmen der Performance-Evaluation kann dann untersucht werden, wie sich die Performance des Gesamtsystems ändert, wenn verschiedene Interface-Konfigurationen an den generischen Komponenten genutzt werden. Bedingt eine Interface-Konfigurationsänderung auch eine Änderung der verwendeten TC der generischen Komponenten, so werden zum Umkonfigurieren des Modells eine Neukomilierung und eventuell neue Adapter notwendig. Dies erschwert und verlangsamt die automatisierte Suche nach einer optimalen Konfiguration.

Für konfigurierbare MMBIF, wie dem OCP oder auch dem PLB, ist der TC-Ansatz zur Interoperabilitätskontrolle zwar machbar, aber mit diversen, oben genannten Problemen verbunden. Die Suche nach einem alternativen Ansatz zur Interoperabilitätskontrolle ist lohnenswert und wird in Abschnitt 4.6 durchgeführt.

Modifiabilities der GP-Elemente

Wie bereits eingangs dieses Abschnittes angedeutet, sind die Regeln zu den Modifiabilities der GP-Elemente für die taktgenaue Modellierung unzureichend. Um dies zu belegen, listet Tabelle 4.11 die Modifiabilities der GP-Grundelemente laut TLM-2.0-LRM auf.

Die Modifiabilities in Tabelle 4.11 sind prinzipiell protokollunabhängig. Die einzige TLM-Phase und somit Protokollabhängigkeit ist mit `BEGIN_RESP` gegeben. Dies kann aber insoweit als Protokoll-unabhängig angesehen werden, dass sich dieses `BEGIN_RESP` auf eine bestimmte TLM-Phase im verwendeten Protokoll bezieht, die vom Target zum Initiator verläuft. `BEGIN_RESP` ist lediglich das BP-Pendant zu einer solchen Phase. Es ist möglich, dass ein Protokoll keine TLM-Phase besitzt, die vom Target zum Initiator verläuft. Das bedeutet, dass es keine Informationen gibt, die vom Target zum Initiatior laufen[16]. Gibt es also keine TLM-Phase vom Target zum Initiator, sind die GP-Grundelemente, die über ein `BEGIN_RESP`-Pendant in ihrer Modifiability gesteuert werden, ohne Bedeutung, da sie aufgrund ihrer sinnvollen Verwendung nur dann im GP-Mapping berücksichtigt werden (siehe Abschnitt 4.3.1), wenn ein Informationsfluss vom Target zum Initiator existiert.

[16] Gibt es Informationen vom Target zum Initiator, gibt es mindestens ein `END_X` oder `BEGIN_Y`, das vom Target zum Initiator verläuft (vgl. Abschnitt 4.3.2 in Zusammenhang mit Anhang B).

Wie bereits in Abschnitt 4.5.1 angedeutet, ist die Beschreibung der Modifiabilities nicht ausreichend, wenn Phasen mehrfach auftreten können, wie z.B. im Rahmen der taktgenauen Modellierung. Während dies für einige Elemente des GPs unkritisch ist, da sie vor Transaktionsbeginn bestimmbar sind und sich während der Transaktion nie ändern (wie z.B. das Kommando oder die Streaming-Width), ist dies bei anderen problematisch. Die Inhalte des Daten-Feldes (`*m_data`) und des Byte-Enable-Feldes (`*m_byte_enable`) müssen bei einer schreibenden Transaktion laut TLM-2.0-LRM vor dem Beginn der eigentlichen Transaktion gefüllt werden. Dies ist bei einer taktgenauen Simulation nicht immer möglich. Wird auch die Funktion (und nicht nur die Kommunikation) taktgenau modelliert, ist pro Busphase immer nur ein einziges Wort der Kommunikation bekannt, und das Daten-Feld der gesamten Transaktion ist erst mit dem letzten Wort des Bursts vollständig gefüllt. Das heißt, die Modifiabilitiy des Daten-Feld-Inhaltes muss einerseits die TLM-Phase beinhalten, die den Datenaustausch symbolisiert (im OCP gibt es dazu z.B. eine eigene Datenphase) und den Wiederholungszähler der entsprechenden Phase berücksichtigen. In Abhängigkeit davon können dann die Teile des Datenfeldes bestimmt werden, die nicht mehr oder noch veränderbar sind und dementsprechend auch die Teile, die zurzeit gelesen werden dürfen. Ein ähnliches Problem existiert für die Modifiability des Byte-Enable-Feldes.

Auch bei der Adresse ist die Modifiability unzureichend erfasst. Diese ist nicht zwingend vor Start (also vor dem ersten `nb_transport_fw`) der Transaktion bekannt. Eine Transaktion kann mit einer Arbitrierungsanfrage an den Bus beginnen, ohne dass eine Adresse benötigt wird. Auch hier muss die Phase benannt werden, mit der die Adresse gültig ist. Darüber hinaus darf die Adresse laut TLM-2.0-LRM von jedem Interconnect genau einmal geändert werden: bei Empfang des ersten `nb_transport_fw`. Ist diese dabei noch nicht bekannt, ist eine Änderung wenig sinnvoll. Auch hier muss die Phase genannt werden, bei der die Adresse geändert werden darf. Darauf aufbauend stellt sich dann die Frage, wie die Modifiability zu deuten ist, wenn die entsprechende Phase mehrfach auftritt. Darf dann jedes mal die Adresse geändert werden?

Letztendlich zeigt Tabelle 4.11 auch, dass es seitens des TLM-2.0-LRM keine Vorgaben, nicht einmal Empfehlungen, bezüglich der Modifiabilities von GP-Erweiterungen gibt. Dies ist bei LT- oder AT-Modellierung auch nicht unbedingt notwendig, da man auf diesen Abstraktionsebenen die Verwendung von GP-Erweiterungen auf das absolute Minimum beschränken sollte. Dies kann auf Kosten der Genauigkeit der Zeitapproximation gehen, erhöht aber sehr stark die Interoperabilität eines solchen Modells. Letztere wiegt auf den abstrakten Modellierungsebenen grundsätzlich schwerer als die zeitliche Genauigkeit, da es dort um den schnellen Aufbau von zeitlich groben Modelllen und um einfachen IP-Austausch geht.

GP-Element		Initiator		Interconnect		Target	
		Setzen	Lesen	Setzen	Lesen	Setzen	Lesen
`m_command`		vor erstem `nb_transport_fw`-Aufruf	immer	nie	immer	nie	immer
`m_address`		vor erstem `nb_transport_fw`-Aufruf	nach Setzen und vor erstem `nb_transport_fw`-Aufruf	vor erstem `nb_transport_fw`-Aufruf	nur bei erstem Empfang von `nb_transport_fw`	nie	nur bei erstem Empfang von `nb_transport_fw`
`m_data`		vor erstem `nb_transport_fw`-Aufruf	immer	nie	immer	nie	immer
`m_length`		vor erstem `nb_transport_fw`-Aufruf	immer	nie	immer	nie	immer
`*m_data`	Write	vor erstem `nb_transport_fw`-Aufruf	immer	nie	immer	nie	immer
	Read	nie	bei/nach `BEGIN_RESP` oder `TLM_COMPLETED`	nie	bei/nach Empfang von `BEGIN_RESP` oder `TLM_COMPLETED`	vor `BEGIN_RESP` oder Rückgabe von `TLM_COMPLETED`	nach Setzen
`m_byte_enable`		vor erstem `nb_transport_fw`-Aufruf	immer	nie	immer	nie	immer
`m_byte_enable_length`		vor erstem `nb_transport_fw`-Aufruf	immer	nie	immer	nie	immer
`*m_byte_enable`		vor erstem `nb_transport_fw`-Aufruf	immer	nie	immer	nie	immer
`m_streaming_width`		vor erstem `nb_transport_fw`-Aufruf	immer	nie	immer	nie	immer
`m_response_status`		vor erstem `nb_transport_fw`-Aufruf auf `TLM_INCOMPLETE_RESPONSE`	bei/nach `BEGIN_RESP` oder `TLM_COMPLETED`	nie	bei/nach Empfang von `BEGIN_RESP` oder `TLM_COMPLETED`	vor `BEGIN_RESP` oder Rückgabe von `TLM_COMPLETED`	nach Setzen
GP-Erweiterungen		Nicht spezifiziert					

Tabelle 4.11.: Modifiabilities der GP-Grundelemente laut TLM-2.0-LRM

Auf der taktgenauen Ebene aber sind GP-Erweiterungen meist unvermeidbar, da nahezu jedes MMBIF Signale enthält, die beim GP-Mapping auf GP-Erweiterungen abgebildet werden und zur Aufrechterhaltung der angestrebten taktgenauen busphasenbasierten Modellierung nicht aus dem Satz der Phasenports entfernt werden können. Dazu gehören z.B. Bus-Lock, Prioritäten, Tags und Informationen, die Auswirkungen auf die simulierte Bearbeitungsdauer haben können, wie die Information ob die Transaktion Daten oder Instruktionen überträgt. Bei komplexen Protokollen wie dem OCP oder AMBA sind dies bis zu 16 GP-Erweiterungen.

Die Art der Verwendung von GP-Erweiterungen hat jedoch Einfluss auf die Simulationsgeschwindigkeit und vor allem auf die potentiell möglichen Fehler im simulierten Verhalten bis hin zu möglichen Simulatorabstürzen. Anhang E erläutert dies im Detail. Im Rahmen der taktgenauen Modellierung müssen also Regeln für GP-Erweiterungen existieren, die folgende Kriterien erfüllen (sortiert nach Wichtigkeit):

1. Sicherstellen einer robusten Simulation, im Sinne von Simulatorabstürzen
2. Bereitstellung eindeutiger Regeln, die bei Befolgung zur sicheren Informationsübertragung mit Hilfe von GP-Erweiterungen führen
3. Gute Simulationsperformance bei Verwendung von GP-Erweiterungen
4. Geringe Anwender-Code-Komplexität

Diese Regeln hängen stark von der Modifiability der Information ab, die mit Hilfe der GP-Erweiterung übermittelt werden sollen. Abschnitt 4.7 wird eine Klassifikation von Modifiabilities einführen und mit ihrer Hilfe die Unzulänglichkeiten in der Definition der Modifiabilities der GP-Grundelemente beheben und die oben geforderten Regeln für GP-Erweiterungen bestimmen.

4.6. Bindungs-Checks

Wie im vorangegangenen Abschnitt beschrieben, sind die Bindungs-Checks zur Kompilierzeit mit Hilfe von TC nicht gut geeignet für die taktgenaue Modellierung, speziell von konfigurierbaren Interfaces. Die Forderung, dass die Verwendung der gleichen TC völlige Interoperabilität garantieren muss, ist aufgrund der in Abschnitt 4.5.2 beschriebenen Probleme zu strikt. Der Mechanismus sollte derart erweitert werden, dass er zuverlässig unzulässige Verbindungen verbietet, alle zulässigen Verbindungen erlaubt, keine Adapter benötigt und auch noch nach dem Kompilieren, zur Unterstützung generischer Komponenten, in seinen Regeln verändert werden kann.

Die Problematik der Bindungs-Checks wurde in Abschnitt 4.5.2 mit Hilfe von GP-Erweiterungen eingeführt. Im Folgenden werden aber auch TLM-Phasen mit in die Bindungs-Checks einbezogen. Dies begründet sich dadurch, dass in konfigurierbaren MMBIFs unter Umständen ganze Signalgruppen optional sind. Stimmen diese Signalgruppen mit den Phasenports einer Busphase überein, bedeutet das, dass diese Busphase und somit auch nach dem

TLM-Phase-Mapping TLM-Phasen aus dem Interface entfallen. Dementsprechend können sich Interfaces eines solchen MMBIF neben den verwendeten GP-Erweiterungen auch in den den verwendeten TLM-Phasen unterscheiden, jedoch ähnlich wie bei den GP-Erweiterungen abhängig davon, ob bei Master oder Slave die entsprechende Phase fehlt, nach wie vor interoperabel sein.

Meine Untersuchungen von konfigurierbaren Protokollen, insbesondere die Arbeit mit dem OCP, legen die Verwendung der in Tabelle 4.12 gezeigten, drei-schichtigen Interoperabilitätsprüfung nahe. Die einzelnen Ebenen werden im Folgenden erläutert. Anschließend wird auf mögliche Implementierungen eingegangen.

Interoperabilitäts-ebene	Überprüfungs-zeitpunkt	Beschreibung
L0	Kompilierzeit	Verwendung einer bestimmten TC legt einerseits den nicht verhandelbaren Satz von nicht ignorierbaren GP-Erweiterungen und TLM-Phasen und andererseits den verhandelbaren fest.
L1	Elaboration	Auf jeder Punkt-zu-Punkt-Verbindung werden zwischen den verbundenen Sockets Verhandlungen über den letztendlich verwendeten Teil der verhandelbaren, nicht ignorierbaren GP-Erweiterungen und TLM-Phasen durchgeführt.
L2	Post-Elaboration	Anwender-definierte, L1-abhängige Interoperabilitäts-Überprüfungen, die über GP-Erweiterungen und TLM-Phasen hinausgehen.

Tabelle 4.12.: Schichten der vorgeschlagenen Interoperabilitätsprüfung

4.6.1. L0-Interoperabilität

Der Mechanismus zur L0-Interoperabilität entspricht dem im TLM-2.0-LRM definierten Bindungs-Check zur Kompilierzeit mit Hilfe von TCs. Dabei wird jedoch die Bedeutung erweitert: Laut TLM-2.0-LRM bedeutet TC-basierende Interoperabilität, dass uneingeschränkt in jedem Fall erfolgreiche Kommunikation möglich ist, beide Kommunikationspartner den gleichen Satz an TLM-Phasen und GP-Erweiterungen kennen[17] und in jedem Fall verwenden[18].

Im Rahmen der taktgenauen Modellierung bedeutet dies zusätzlich, dass über diesen unverhandelbaren Satz von GP-Erweiterungen und TLM-Phasen hinaus beide Kommunikationspartner einen weiteren Satz von GP-Erweiterungen und TLM-Phasen kennen können.

[17] Das heißt, dem Modellentwickler und dem Compiler ist der Typname der Erweiterung bzw. TLM-Phase bekannt.

[18] Das heißt, der Code des Modells bearbeitet die GP-Erweiterung bzw. TLM-Phase, bei jeder möglichen Kommunikation.

Jedoch müssen diese GP-Erweiterungen oder TLM-Phasen nicht in jedem Fall verwendet werden[19].

Ein Beispiel für eine Erweiterung aus solch einem Satz ist die GP-Erweiterung zur Signalisierung von Bursts mit unbekannter Länge, aus dem in Abschnitt 4.5.2 in den Abbildungen 4.9 und 4.10 auf Seite 57 eingeführten Beispiel. Master M1 und S1 kennen und verwenden die Erweiterung, während Master M2 und Slave S2 die Erweiterung lediglich kennen (aufgrund der Tatsache, dass sie das OCP kennen), aber vorsätzlich in keinem Fall verwenden.

GP-Erweiterungen und TLM-Phasen, die in keinem der beiden Sätze existieren und dennoch verwendet werden, müssen im Sinne des TLM-2.0-LRM ignorierbar sein.

Der unverhandelbare Satz von GP-Erweiterungen und TLM-Phasen ergibt sich während des GP- und TLM-Phase-Mappings. Eine GP-Erweiterung gehört zu diesem Satz, wenn für jeden Master und jeden Slave der Peripherie eines Kommunikationsautomaten mindestens eine Busphase mit einem Port existiert, der beim GP-Mapping auf die entsprechende GP-Erweiterung abgebildet wird. Eine TLM-Phase gehört zu diesem Satz, wenn für jeden Master und jeden Slave der Peripherie eines Kommunikationsautomaten eine Busphase existiert, von der ein Rand auf die entsprechende TLM-Phase abgebildet wird. Erfordert das MMBIF des Kommunikationsautomaten mehrfaches GP- und TLM-Phase-Mapping (siehe Abschnitt 4.3.3, Absatz „Mehrfaches GP- und TLM-Phase-Mapping“), so muss diese Anforderung für alle durchgeführten Mappings gelten.

Der Satz der verhandelbaren GP-Erweiterungen und TLM-Phasen ergibt sich nach dem (mehrfachen) GP- und TLM-Phase-Mapping, indem man den unverhandelbaren Satz von GP-Erweiterungen und TLM-Phasen vom gesamten Satz von GP-Erweiterungen und TLM-Phasen abzieht.

4.6.2. L1-Interoperabilität

Nach Abschluss der Konstruktion des Modells und vor dem Start der Simulation untersucht SystemC in der sog. Elaboration die Integrität des Modells. Zusätzliche Untersuchungen können in verschiedenen Funktionen, die vom Simulator aufgerufen werden, implementiert werden. TLM-Sockets können in einer solchen Funktion Informationen über die aus dem verhandelbaren Satz der GP-Erweiterungen und TLM-Phasen tatsächlich genutzten Erweiterungen und Phasen austauschen und mit Hilfe dieser Informationen feststellen, ob die Sockets interoperabel sind. Es handelt sich also um eine Überprüfung zur Laufzeit.

Die Information, ob eine GP-Erweiterung oder TLM-Phase benutzt wird, reicht allein nicht zum Überprüfen der Interoperabilität aus. Man betrachte dazu ein weiteres Mal das Beispiel aus Abschnitt 4.5.2 (Abbildungen 4.9 auf Seite 56 und 4.10 auf Seite 57). Slave S1 benutzt die GP-Erweiterung zur Signalisierung eines Bursts unbekannter Länge, Master M2 nutzt sie nicht. Trotzdem sind sie interoperabel. Im Gegensatz dazu sind M1, der die Erweiterung nutzt, und S2, der sie nicht nutzt, nicht interoperabel. Aus diesem Grund führe ich die

[19] Das heißt, der Code des Modells geht davon aus, dass die Erweiterung immer, nie oder nur in speziellen Fällen verwendet wird.

Erfordernisgrade verhandelbarer GP-Erweiterungen bzw. TLM-Phasen wie in Tabelle 4.14 ein.

Erfordernisgrad	Beschreibung
zwingend	Das Modul erfordert vom Kommunikationpartner, dass die GP-Erweiterung bzw. TLM-Phase in jedem Fall verwendet wird.
optional	Das Modul stellt keine Anforderungen an die Verwendung der GP-Erweiterung bzw. TLM-Phase auf Seite des Kommunikationspartners.
verboten	Das Modul erfordert vom Kommunikationpartner, dass die GP-Erweiterung bzw. TLM-Phase unter keinen Umständen verwendet wird.

Tabelle 4.13.: Erfordernisgrade von verhandelbaren GP-Erweiterungen und TLM-Phasen

Jedes Modul kann nun für jede GP-Erweiterung und jede TLM-Phase aus dem verhandelbaren Satz einer TC für jeden Socket einen Erfordernisgrad bestimmen. Werden nun zwei Sockets verbunden, werden während der Elaboration die Erfordernisgrade der GP-Erweiterungen und TLM-Phasen paarweise verglichen. Das Ergebnis des Vergleichs ist entweder die Feststellung, dass keine L1-Interoperabilität vorliegt, oder aber ein resultierender Erfordernisgrad. Tabelle 4.14 zeigt, welche Erfordernisgrade L1-interoperabel sind und welche resultierenden Erfordernisgrade sich dann ergeben. Ein „✘“ markiert dabei nicht L1-interoperable Erfordernisgrade.

	zwingend	optional	verboten
zwingend	zwingend	zwingend	✘
optional	zwingend	optional	verboten
verboten	✘	verboten	verboten

Tabelle 4.14.: L1-Interoperabilität der verschiedenen Erfordernisgrade

Mit Hilfe dieser Erfordernisgrade kann die Bindungsproblematik, die in Abschnitt 4.5.2 in den Abbildungen 4.9 und 4.10 erläutert wurde, wie in Abbildung 4.15 gelöst werden. Man erkennt, dass die zulässigen Verbindungen L1-interoperabel sind, während die unzulässige Verbindung bei der L1-Interoperabilitätsüberprüfung als nicht interoperabel erkannt wird. Es werden keinerlei Adapter benötigt.

Allgemeingültige Regeln zur Bestimmung der Erfordernisgrade sollen hier nicht festgelegt werden, da diese zu stark vom zu modellierenden Protokoll abhängen. Als Faustregel kann festgehalten werden, dass bei Nichtexistenz des Signals im Interface die GP-Erweiterung, auf die das Signal abgebildet wird, als verboten markiert wird. Das Modell eines Signaltreibers sollte die GP-Erweiterung als zwingend identifizieren, da er darüber eine aus seiner Sicht wichtige Information mitteilen will. Eine Signalsenke hingegen wird die GP-Erweiterung als optional markieren, wenn im Falle eines fehlenden Treibers ein sicherer Defaultwert existiert. Existiert kein sicherer Defaultwert, sollte die Erweiterung auf zwingend gesetzt werden. Diese

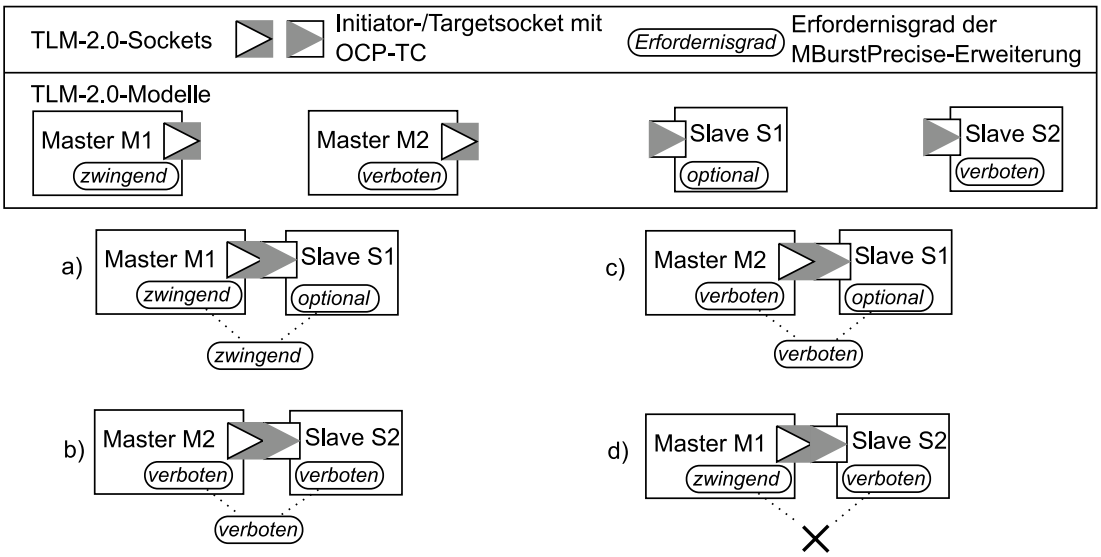


Abbildung 4.15.: Ergebnisse des L1-Interoperabilitätschecks für die MBurstPrecise-GP-Erweiterung

Regelungen werden im Beispiel in den Abbildungen 4.9 bzw. 4.15 verwendet. Das Beispiel in Abbildung 4.17 auf Seite 67 hingegen weicht von der Faustregel ab.

Auch generische Komponenten können mit diesem Ansatz realisiert werden. Abbildung 4.16 zeigt einen generischen Master bezüglich der MBurstPrecise-Erweiterung, der abhängig vom angeschlossenen Slave Bursts unbekannter Länge absetzt. Die Slaves S1 und S2 sind aus dem bereits bekannten Beispiel, dazu kommt (der eher theoretische) Slave S3, der ausschließlich Bursts unbekannter Länge bearbeiten kann und dementsprechend den Erfordernisgrad der Erweiterung auf zwingend setzt.

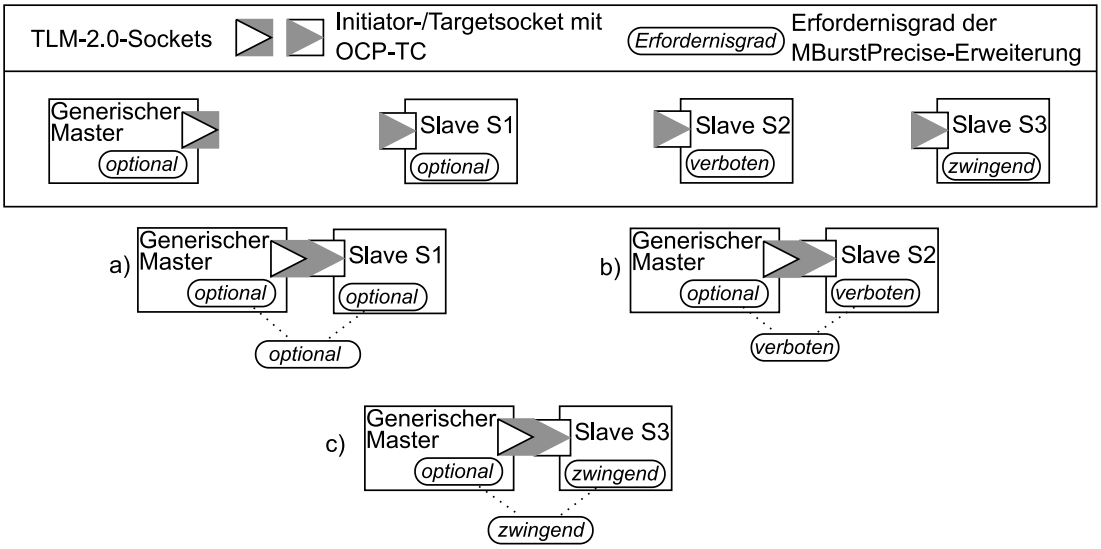


Abbildung 4.16.: Generische Komponenten und L1-Interoperabilität

Ist der generische Master mit Slave S2 verbunden (Abbildung 4.16b), so erkennt der Master am resultierenden Erfordernisgrad, dass der Slave keine Bursts unbekannter Länge un-

terstützt und kann sich daran anpassen. Bei der Verbindung mit Slave S3 (Abbildung 4.16c) kann der generische Master erkennen, dass er die GP-Erweiterung verwenden muss. Ist er hingegen mit Slave S1 verbunden (Abbildung 4.16a), so zeigt der resultierende Erfordernisgrad an, dass es dem Master freisteht zu entscheiden, ob die Erweiterung eingesetzt wird. Dies bedeutet aber auch, dass der Slave S1 zur Laufzeit erkennen muss, ob die Erweiterung benutzt wird, da er am resultierenden Erfordernisgrad lediglich erkennen kann, dass es dem Master nun freisteht zu entscheiden. Ein Mechanismus, um zur Laufzeit die Verwendung einer GP-Erweiterung zu erkennen, wird in Abschnitt 4.8 erläutert.

4.6.3. L2-Interoperabilität

Nachdem die L1-Interoperabilität überprüft wurde, das heißt, wenn die verbundenen Sockets interoperabel bezüglich ihrer GP-Erweiterungen und TLM-Phasen sind, sollte dem Nutzer der Sockets die Möglichkeit gegeben werden, die Legalität einer Verbindung einer weitergehenden Prüfung zu unterziehen. Dabei können Daten abgeglichen werden, die nicht bei der L1-Interoberabilität berücksichtigt wurden. Dazu gehören z.B. Bitbreiten-Vergleiche der modellierten Ports. Als Beispiel soll das ARID-Signal des AMBA-AXI-Interfaces dienen. Dieses Signal erlaubt sog. Out-Of-Order-Transaktionen, d.h. Responses von Transaktionen können in anderer Reihenfolge empfangen werden, als ihre Requests gesendet wurden. Um dann also die Response einer Transaktion dem Request zuzuordnen, müssen Request und Response die gleiche ID haben. Das ARID-Signal überträgt dabei die ID des Requests, RID ist die ID der Response.

Man betrachte den linken Teil in Abbildung 4.17 auf der nächsten Seite. Dort sind verschiedene AXI-Verbindungen skizziert, die (A)RID-Signale sind explizit eingezeichnet. Beim GP-Mapping wird das ARID-Signal eines jeden Moduls auf die gleiche GP-Erweiterung abgebildet. Dabei wird als Datentyp zur Speicherung der ID in der GP-Erweiterung `uint8_t` gewählt werden, da die breiteste ID mit 8 Bit gegeben ist. Abbildung 4.17 zeigt rechts die Erfordernisgrade der Erweiterung. Man erkennt, dass dieser Grad nie auf zwingend gesetzt ist, da, wie in Abbildung 4.17 links zu sehen, das AXI-Protokoll für jeden Fall, egal ob Master, Slave, keiner oder beide die (A)RID verwenden, eine Bindungslösung definiert. Folglich sind die Bindungen auch im TLM-Modell zulässig[20].

Besonders zu beachten ist hier die Verbindung von M5 und S5 in Abbildung 4.17. Diese Verbindung ist nicht zulässig, da verschiedene 8-Bit-Request-IDs vom Master (z.B. `0x2e` und `0x3e`) vom Slave auf die gleiche 4-Bit-Response-IDs (bei vorangegangenem Beispiel `0xe`) abgebildet werden würden und so der Master falsche Response-Request-Zuordnungen treffen

[20] Der Vollständigkeit halber hier eine kurze Skizze des Verhaltens in TLM: Ähnlich wie in RTL wird bei Abwesenheit der GP-Erweiterung im Master aber Anwesenheit im Slave, wenn also der Slave erkennt, dass der resultierende Erfordernisgrad auf verboten gesetzt ist, der Slave so arbeiten, als wäre die ID=0 für alle Transaktionen. Verwendet der Master die Erweiterung, aber der Slave nicht, so kann der Master feststellen, ob der resultierende Erfordernisgrad der RID-Erweiterung auf verboten gesetzt ist und dann einfach den Wert der ARID-Erweiterung als Wert für RID ansehen.

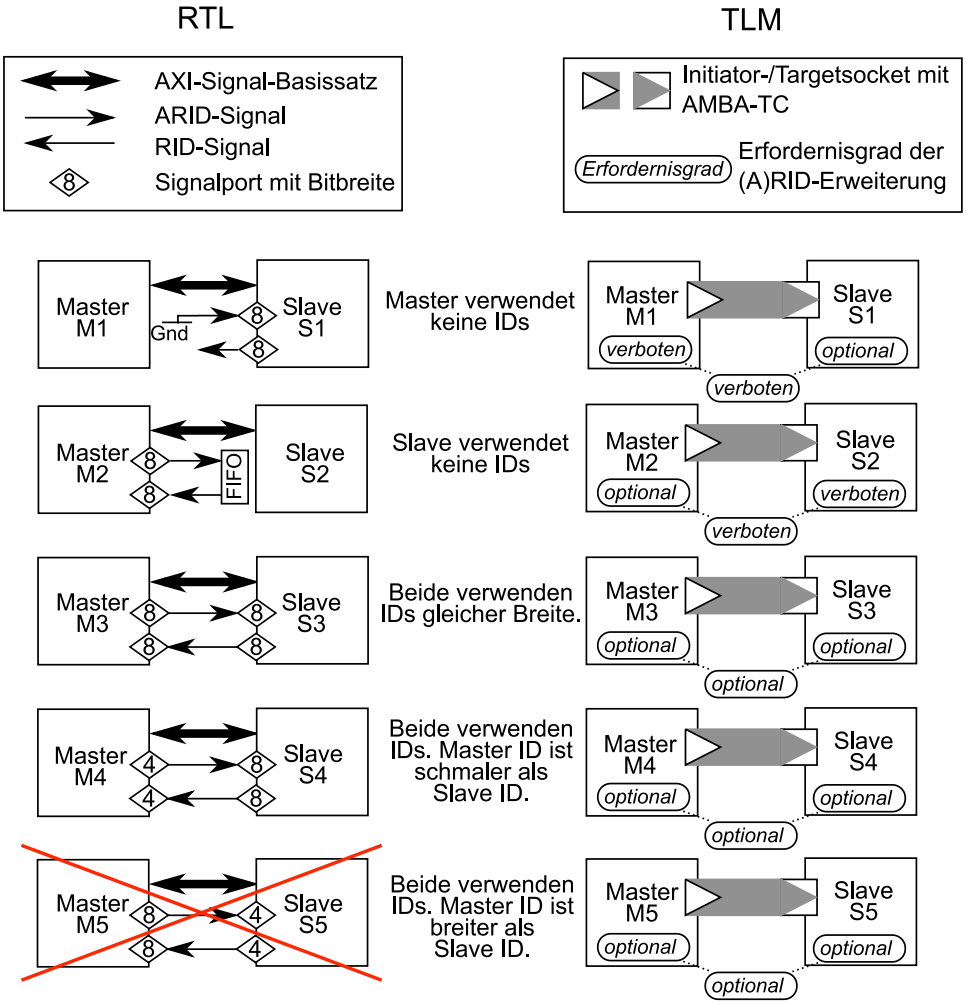


Abbildung 4.17.: Beispiel für die Grenzen der L1-Interoperabilitätsprüfung

würde. Man erkennt aber auch, dass die entsprechenden Sockets L1-interoperabel sind, sie verwenden beide die gleiche GP-Erweiterung. Der L1-Interoperabilitäts-Test ist aber nicht in der Lage auszuwerten, ob die genutzten Bits der Erweiterung auch übereinstimmen.

Es ist vorstellbar, in einem Socket neben dem Erfordernisgrad einer GP-Erweiterung auch deren genutzte Bitmaske zu speichern und in der L1-Interoperabilität zu berücksichtigen. Jedoch wäre dies nicht ausreichend. In einigen Protokollen (wie bei AMBA) ist die beschriebene Situation fatal, während sie bei anderen Protokollen durch weitere Mechanismen abgefangen werden kann. Man müsste somit auch den Bitbreiten verschiedene Erfordernisgrade hinzufügen. Darüber hinaus gibt es gelegentlich auch Abhängigkeiten zwischen GP-Erweiterungen, während die L1-Interoperabilität immer nur eine GP-Erweiterung betrachtet. Alle denkbaren Tests, die über den beschriebenen L1-Test hinausgehen, über einen generischen Mechanismus abzuwickeln, wäre sehr aufwändig. Darüber hinaus sind die Fälle, in denen L0- und L1-Interoperabilitätschecks nicht reichen, selten und rechtfertigen kaum eine komplexe Standardisierung.

Aus diesem Grund sollte der Nutzer, wie bereits eingangs erwähnt, nach der L1-Prüfung die Möglichkeit haben, solche komplexen Tests selbst durchzuführen. Diese Überprüfung

nenne ich dann L2-Interoperabilität. Dazu soll der Nutzer eines TLM-2.0-Socket einen Zeiger auf ein polymorphes Objekt, der von der Gegenseite festgelegt wird, erhalten. Über diesen Zeiger kann dann der Nutzer des TLM-2.0-Sockets mit Hilfe dynamischer Casts oder anderer geeigneter Mechanismen die Information erhalten, die er für seine Tests braucht. Dabei darf er aber nicht mehr die Erfordernisgrade von GP-Erweiterungen oder TLM-Phasen verändern.

Im obigen Beispiel würde der Nutzer die Bitbreiten der modellierten ID-Signale vergleichen und bei der Bindung von M5 und S5 mit einer geeigneten Fehlermeldung die Simulation beenden.

4.6.4. Bindungschecks bezüglich der GP-Grundelemente

Die bisherigen Abschnitte haben ausschließlich GP-Erweiterungen und TLM-Phasen diskutiert. In konfigurierbaren MMBIF ist es aber durchaus möglich, Signale, die im GP-Mapping auf GP-Grundelemente gemappt werden, abhängig von der Konfiguration des MMBIF zu verwenden oder nicht. Ist dies der Fall, können Pseudo-GP-Erweiterungen definiert werden. Diese Erweiterungen werden dann genutzt, um ihnen für die L1-Interoperabilitätsprüfung Erfordernisgrade zuzuweisen. Eine andere Möglichkeit ist, solche Überprüfungen in der L2-Ebene abzudecken.

4.6.5. Integration in TLM-2.0

Die Implementierung für die L0-Interoperabilität ist in TLM-2.0 bereits mit der TC-basierten Interoperabilität gegeben. Entscheidend ist also die Frage, wie L1- und L2-Interoperabilität und deren Überprüfungen implementiert werden können. Zuerst muss die Frage diskutiert werden, wie grundsätzlich der Austausch der für die L1- und L2-Überprüfungenen notwendigen Informationen erfolgen kann, danach werden die Datenstrukturen für diese Informationen festgelegt.

Informationsaustausch

Für den Austausch der Informationen stehen zwei Optionen zur Diskussion. Entweder werden neue IMCs zum TLM-2.0-Interface hinzugefügt, oder einer der bereits vorhandenen IMC kann weiter verwendet werden. Die auszutauschenden Informationen beinhalten die Zuordnung von Erfordernisgraden zu den GP-Erweiterungen und TLM-Phasen des verhandelbaren Satzes und den in Abschnitt 4.6.3 erwähnten Zeiger für die L2-Tests. Der Austausch einer komplexen Datenstruktur ist folglich angebracht. Die einzige Möglichkeit, dies mit Hilfe der existierenden IMCs zu bewerkstelligen, ist eine GP-Erweiterung zu definieren, die die benötigten Informationen überträgt. Dann können NBTI, BTI oder DTI (siehe Abschnitt 2.3.2) zum Übertragen der Information eingesetzt werden. Jedoch entspricht der Austausch von Bindungscheck-Informationen keinem der laut TLM-2.0 angedachten Zwecke der vorhandenen Interfaces. Es ist nicht ratsam, eines der existierenden Interfaces zu verwenden,

da die damit verbundene Doppeldeutigkeit Fehlverwendungen des entsprechenden Interfaces zuträglich ist und auch das Erlernen der damit komplexeren Regeln für das überladene Interface schwieriger ist. In dieser Arbeit wird also TLM-2.0 um ein neues Interface zur Übertragung der L1- und L2-Information erweitert.

Die TLM-2.0-Interfaces sind in ihren Datentypen nicht festgelegt, also nicht auf GP und TLM-Phase fixiert. Die Problematik der L1- und L2-Bindungstests wurde hier anhand von GP-Erweiterungen und TLM-Phasen eingeführt, kann aber grundsätzlich auch für andere erweiterbare Datentypen oder völlig anders geartete dynamische, also nicht zur Kompilierzeit bestimmte, Bindungstests eingesetzt werden[21]. Im Folgenden wird also erst ein generisches Interface zur Bindungsüberprüfung zur Simulationslaufzeit eingeführt und anschließend wird dieses Interface für die GP- und TLM-Phasen-abhängigen L1- und L2-Tests verwendet.

Interfaceerweiterung

Der erste Schritt zur Interfaceerweiterung ist die Definition des Datentyps, der die Information tragen soll, die für die Bindungschecks zur Laufzeit verwendet wird. Wie oben erläutert, darf dieser Datentyp keine Annahmen über den Payload- und Phasen-Datentyp machen. Aus diesem Grund definiere ich dafür eine abstrakte Basisklasse, die dann protokollabhängig implementiert werden kann. Abbildung 4.18 zeigt das Klassendiagramm dazu. Man erkennt, dass TLM-2.0 um die abstrakte Klasse `tlm_rt_bind_config_base` erweitert wird. Ein Konfigurationsobjekt für die Laufzeit-Bindungschecks muss dann von `tlm_rt_bind_config_base` abgeleitet werden und dementsprechend `check_against` und `clone` implementieren.

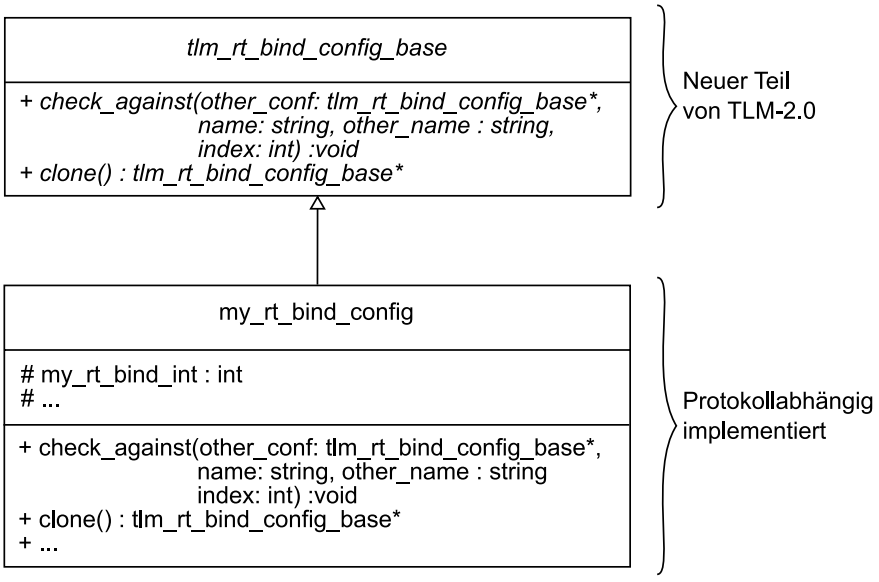


Abbildung 4.18.: Klassendiagramm für die L1- und L2-Basisklassen

Die Bedeutung dieser Funktionen ist wie folgt:

[21] Eine genaue Erörterung der über MMBIF hinaus gehenden Möglichkeiten der L1- und L2-Bindungstests geht aber über den Rahmen der Arbeit hinaus.

`check_against` : Diese Funktion vergleicht die übergebene Konfiguration `other_conf` mit der Konfiguration auf der `check_against` aufgerufen wird. Sollte bei diesem Vergleich eine Interoperabilitätsverletzung festgestellt werden, so soll dies mit Hilfe von `SC_REPORT_ERROR` (siehe [IEEE06b]) gemeldet werden. Ergibt sich aus dem Vergleich eine resultierende Konfiguration (vgl. Abschnitt 4.6.2), so entspricht die Konfiguration auf der `check_against` aufgerufen wurde nach dem Aufruf von `check_against` der resultierenden Konfiguration.

Zur Erzeugung aussagekräftiger Meldungen werden der Funktion der Name des SystemC-Objektes (in der Regel ein TLM-2.0-Socket), das die Konfiguration besitzt, auf der `check_against` aufgerufen wird, und der Name des SystemC-Objekts, das `other_conf` besitzt, übergeben. Da TLM-Sockets auch mehrfach gebunden werden können, wird der Funktion auch der Index der Bindung übergeben. Damit kann bei einem mehrfach gebundenen Socket identifiziert werden, bei welcher Bindung der Fehler auftrat.

`clone` : Diese Funktion soll derart implementiert werden, dass über die abstrakte Basisklasse `tlm_rt_bind_config_base` eine Kopie der Subklasse angelegt werden kann.

Mit Hilfe dieser Funktionen kann der Laufzeit-Bindungscheck komplett in den TLM-2.0-Sockets abgewickelt werden, ohne dass diese exakte Kenntnisse von der Natur des Checks haben (siehe weiter unten).

Die zum Austausch der Konfigurations-Objekte verwendeten Interfaces `tlm_fw_rt_bind_check` und `tlm_bw_rt_bind_check` und deren Positionen in der Klassenhierarchie der TLM-2.0-Interfaces sind in Abbildung 4.19 auf der nächsten Seite dargestellt. Vergleichbar zum NBTI existiert das Interface für Laufzeit-Bindungschecks in einer forward- und einer backward-Variante, damit ein Interconnect unterscheiden kann, von welcher Richtung eine Anfrage nach der Konfiguration empfangen wurde.

Die IMCs des L1-Interfaces haben folgende Funktionen (`_fw-` bzw. `_bw`-Präfix wird vorsätzlich nicht genannt, da für beide Variante die gleiche Erläuterung zutrifft):

`tlm_rt_bind_config_base* get_config()` : Diese Funktion liefert die Konfiguration der Klasse (in der Regel ein TLM-2.0-Socket), die das `tlm_fw_transport_if` bzw. `tlm_bw_transport_if` implementiert. Die Konfiguration wird dabei als Zeiger auf die Basisklasse übertragen. Diese kann dann zur Interoperabilitätsprüfung herangezogen werden.

`std::string get_name()` : Diese Funktion liefert den Instanznamen der Klasse, die das `tlm_fw_transport_if` bzw. `tlm_bw_transport_if` implementiert. Dies ist in der Regel der Name eines TLM-2.0-Sockets oder eines SystemC-Moduls. Der Name wird zur Erzeugung von Fehlermeldungen im Falle von L1-Interoperabilitätsverletzungen benötigt.

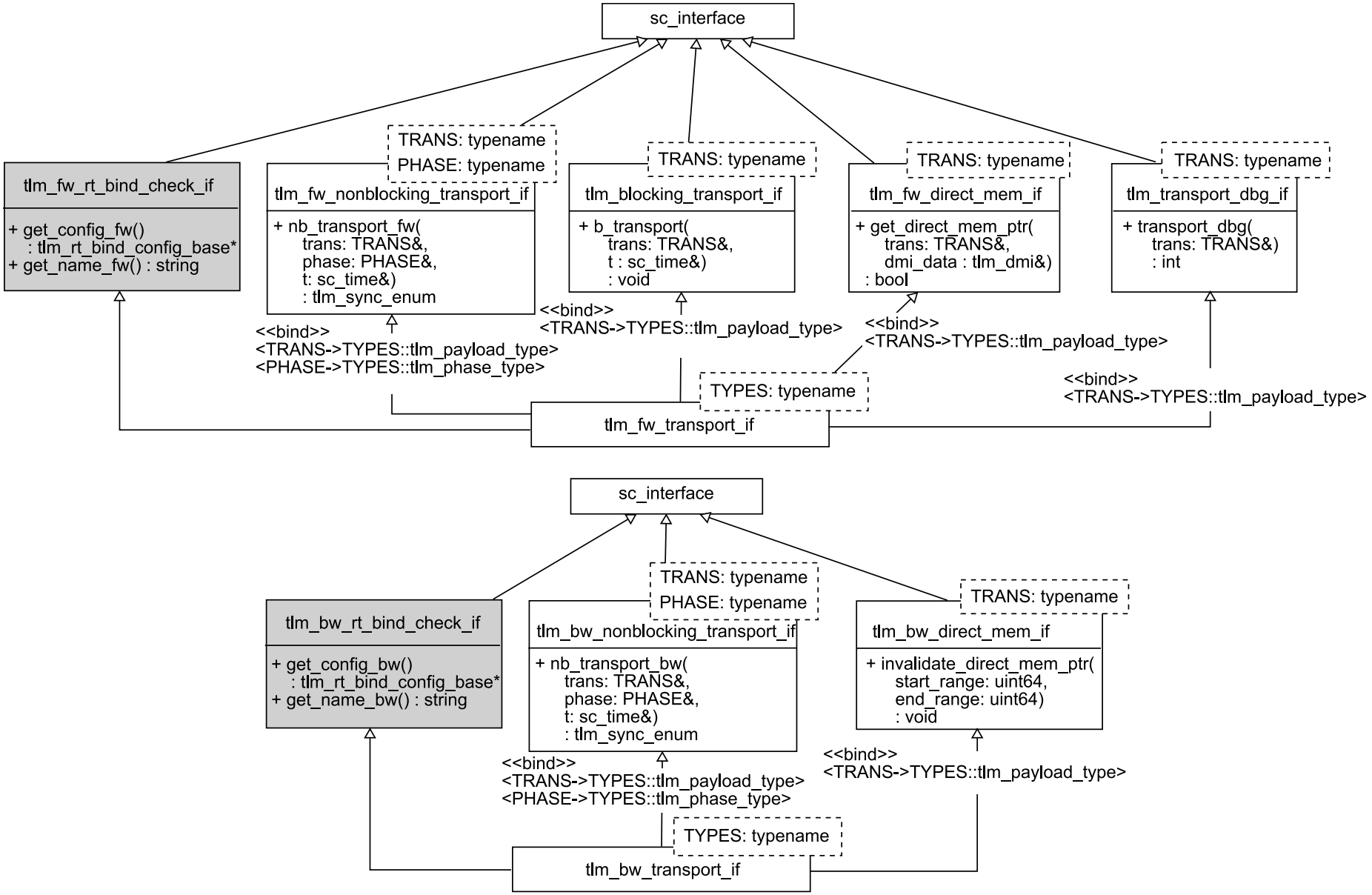


Abbildung 4.19.: Erweiterung der TLM-2.0-Interfaces für die L1-Interoperabilitätsprüfung

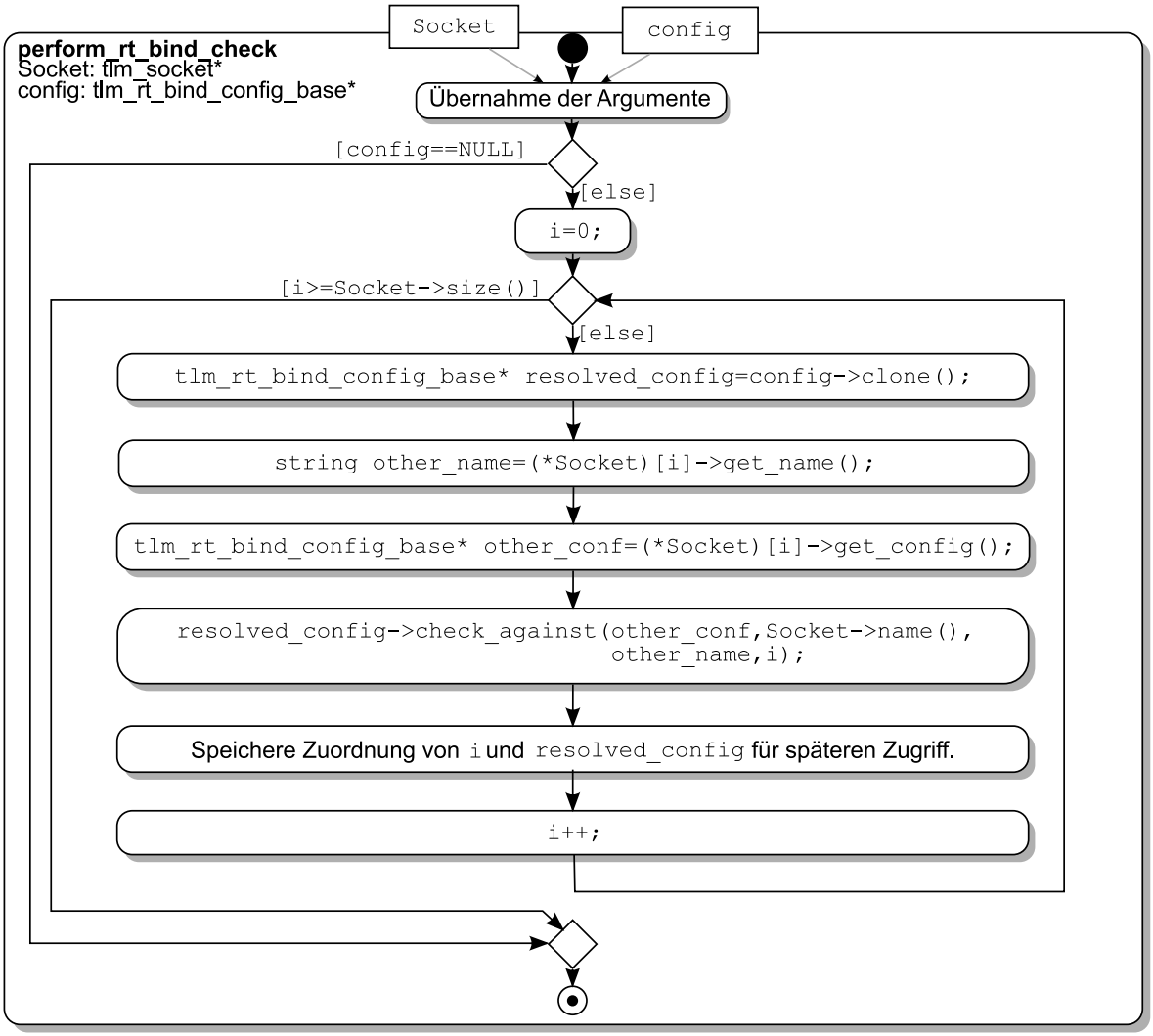


Abbildung 4.20.: Ablauf des Laufzeit-Bindungschecks (`_fw`- und `_bw`-Zusätze an den IMCs sind absichtlich nicht genannt worden. Wenn `Socket` ein Initiatorsocket ist, ist `_fw` zu verwenden. Ist `Socket` ein Targetsocket, ist `_bw` zu verwenden.)

Laufzeit-Bindungscheck und dessen Integration in die TLM-2.0-Sockets

Der mit Hilfe des Interfaces für Laufzeit-Bindungschecks ermöglichte Bindungscheck ist in Abbildung 4.20 dargestellt.

Zur Überprüfung der Bindungen eines Sockets werden ein Zeiger `Socket` auf diesen Socket und ein Zeiger auf dessen Konfiguration `config` benötigt. Besitzt der Socket keine Konfiguration, so wird kein Bindungstest durchgeführt. Die Annahme dabei ist, dass ein Socket ohne Konfiguration stets mit einem Socket ohne Konfiguration kompatibel ist. Sobald einer der verbundenen Sockets eine Konfiguration hat, wird mindestens ein Bindungstest durchgeführt, da beide Sockets die Bindung zu überprüfen versuchen.

Beim Test einer Bindung wird zuerst die Konfiguration des Sockets kopiert, da `check_against`, wie oben erläutert, die Konfiguration ändern kann, auf der es aufgerufen wird. Die initiale Konfiguration des Sockets muss aber erhalten bleiben, damit sie für alle Ein-

zelbindungstests verwendet werden kann. Nach Abruf vom Namen und der Konfiguration eines verbundenen Sockets wird `check_against` aufgerufen. Danach werden die resultierende Konfiguration und der Index der Bindung geeignet gespeichert, sodass der Nutzer eines Sockets später die resultierende Konfiguration einer Bindung untersuchen kann.

Dieser Laufzeit-Bindungscheck soll automatisch von verbundenen TLM-Sockets durchgeführt werden. Aus diesem Grund werden die TLM-2.0-Sockets `tlm_initiator_socket` und `tlm_target_socket` wie in Abbildung 4.22 auf der nächsten Seite gezeigt erweitert. Der Initiator- und der Targetsocket werden dabei von einer neuen Klassen `tlm_rt_bind_socket_attachment` abgeleitet. Diese Klasse enthält eine Funktion `perform_rt_bind_check`, welche den in Abbildung 4.20 beschriebenen Test durchführt. Dabei wird der für den Test benötigte Zeiger auf den Socket als Argument übergeben. Der Zeiger auf die Konfiguration des Sockets `config` aus Abbildung 4.20 ist ein Attribut der Klasse. Dieses Attribut kann vom Nutzer des Sockets über `set_rt_bind_config_ptr` gesetzt werden. Da es sich bei dem Attribut um einen Zeiger handelt, muss das Konfigurationsobjekt ausserhalb des Sockets alloziert werden. Das übergebene Konfigurationsobjekt darf anschliessend nur dann de-alloziert werden, wenn anschliessend oder direkt davor der Socket auch de-alloziert wird. In der Regel ist dies am Ende der Simulation der Fall.

Die Signatur von `set_rt_bind_config_ptr` zeigt, dass das Konfigurationsobjekt nicht als Zeiger auf `tlm_rt_bind_config_base`, sondern als Zeiger auf `TYPES::tlm_rt_bind_config_type` übergeben wird. Dies bedeutet, dass die TC nunmehr neben dem Typ des Payloads und der Phase (siehe Abschnitt 2.3.2) auch den Typ der Laufzeit-Bindungskonfiguration festlegt. Dieses Vorgehen stellt sicher, dass Sockets mit gleicher TC nicht unterschiedliche Konfigurationsobjekte nutzen können. Listing 4.21 zeigt ein Beispiel für die Struktur einer TC im erweiterten TLM-2.0 (vgl. Listing 2.9). Die Voraussetzung, die `my_rt_bind_config` in Listing 4.21 erfüllen muss, ist, dass `my_rt_bind_config` von `tlm_rt_bind_config_base` abgeleitet ist.

```
1 struct my_tlm2_types_class
2 {
3   typedef my_payload_type   tlm_payload_type;
4   typedef my_phase_type     tlm_phase_type;
5   typedef my_rt_bind_config tlm_rt_bind_config_type;
6 };
```

Listing 4.21: Beispiel für eine Types Class im erweiterten TLM-2.0

Die Funktion `perform_rt_bind_check` wird von der Funktion `end_of_elaboration` der Sockets aufgerufen, welche automatisch vom SystemC-Kernel vor dem Start der Simulation aufgerufem wird (siehe [IEEE06b]). Dementsprechend werden die Laufzeit-Bindungstest automatisch ausgeführt.

Die Funktion `get_resolved_config` der Klasse `tlm_rt_bind_socket_attachment` kann nach der Durchführung der Laufzeit-Bindungstests, also z.B. während der Simulation, verwendet werden, um die resultierenden Konfigurationen der Bindungen eines Sockets zu erhalten. Dies sind die Konfigurationen, die in der vorletzten Aktivität in Abbildung 4.20 gespeichert wurden.

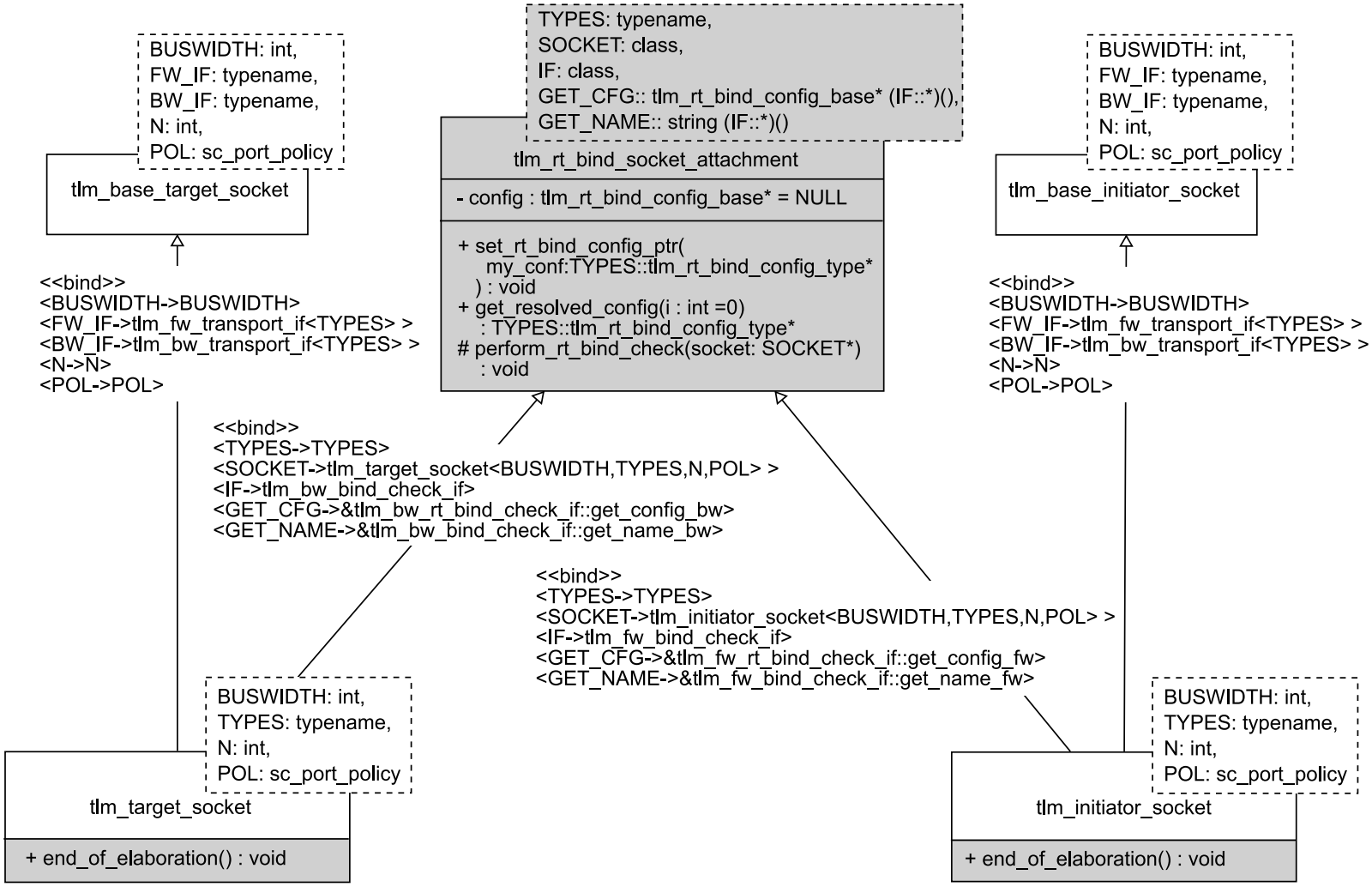


Abbildung 4.22.: Erweiterung der TLM-2.0-Sockets für automatische Laufzeit-Bindungschecks

Erweiterung der Base-Protocol-Types-Class

Die in den vorangegangenen Abschnitten erläuterte TLM-2.0-Interfaceerweiterung für Bindungschecks zur Laufzeit bedingt eine neue Typdefinition `tlm_rt_bind_config_type` in TCs (siehe Listing 4.21). Folglich muss auch die TC des Base-Protocol (BP) erweitert werden. Das BP ist für die AT- und LT-Modellierung vorgesehen und unterstützt keinerlei Verhandlungen über Erweiterungen und Phasen; der verhandelbare Satz von Erweiterungen und Phasen ist also leer. Dies sollte auch in der TC deutlich werden. Listing 4.23 zeigt die entsprechend veränderte TC des BP. Mit dieser Änderung verwenden BP-Module eine Konfiguration für die Laufzeitbindungstests, die keinerlei Informationen enthält und deren Bindungstest nie fehlschlagen kann. Dies reduziert den Bindungstest auf die L0-Ebene (siehe Abschnitt 4.6), also auf Bindungstests zur Kompilierzeit. Somit haben die gezeigten Erweiterungen von TLM-2.0 keinen Einfluss auf BP-Modelle.

```
1  struct tlm_base_protocol_types
2  {
3    struct tlm_no_rt_negotiations_allowed : public tlm_rt_bind_config_base
4    {
5      void check_against(const tlm_rt_bind_config_base* other_conf,
6                         const std::string& name,
7                         const std::string& other_name,
8                         unsigned int index)
9      {}
10     tlm_rt_bind_config_base* clone() const
11     {return NULL;}
12   };
13   typedef tlm_generic_payload tlm_payload_type;
14   typedef tlm_phase tlm_phase_type;
15   typedef tlm_no_rt_negotiations_allowed tlm_rt_bind_config_type;
16 };
```

Listing 4.23: Types-Class des Base-Protocol im erweiterten TLM-2.0

Laufzeit-Bindungstest-Konfiguration für L1- und L2-Tests

Mit Hilfe der oben beschriebenen Laufzeit-Bindungstest können nun auch die L1- und L2-Bindungstests wie in den Abschnitten 4.6.2 und 4.6.3 beschrieben, umgesetzt werden.

Abbildung 4.24 auf der nächsten Seite zeigt das Klassendiagramm für die **L1-Konfiguration** `L1_config`, die Datenstruktur, die zur Speicherung der Erfordernisgrade der GP-Erweiterungen und TLM-Phasen verwendet werden kann. Sie enthält je zwei STL-Maps [Josu99], die GP-Erweiterungen bzw. TLM-Phasen Erfordernisgrade zuordnen. Da sowohl GP-Erweiterungen (siehe Anhang E) als auch TLM-Phasen (siehe TLM-2.0-LRM) eindeutige Integer-IDs haben, kann der Schlüssel einer solchen Map ein Integer sein[22].

Zum Setzen bzw. Auslesen der Erfordernisgrade der GP-Erweiterungen und TLM-Phasen gibt es je eine Funktion. Um nicht jeder GP-Erweiterung oder TLM-Phase aus dem verhandelbaren Satz eines Protokolls explizit einen Erfordernisgrad zuordnen zu müssen, ist ein

[22]Dies erlaubt die Implementierung der Map als Vektor.

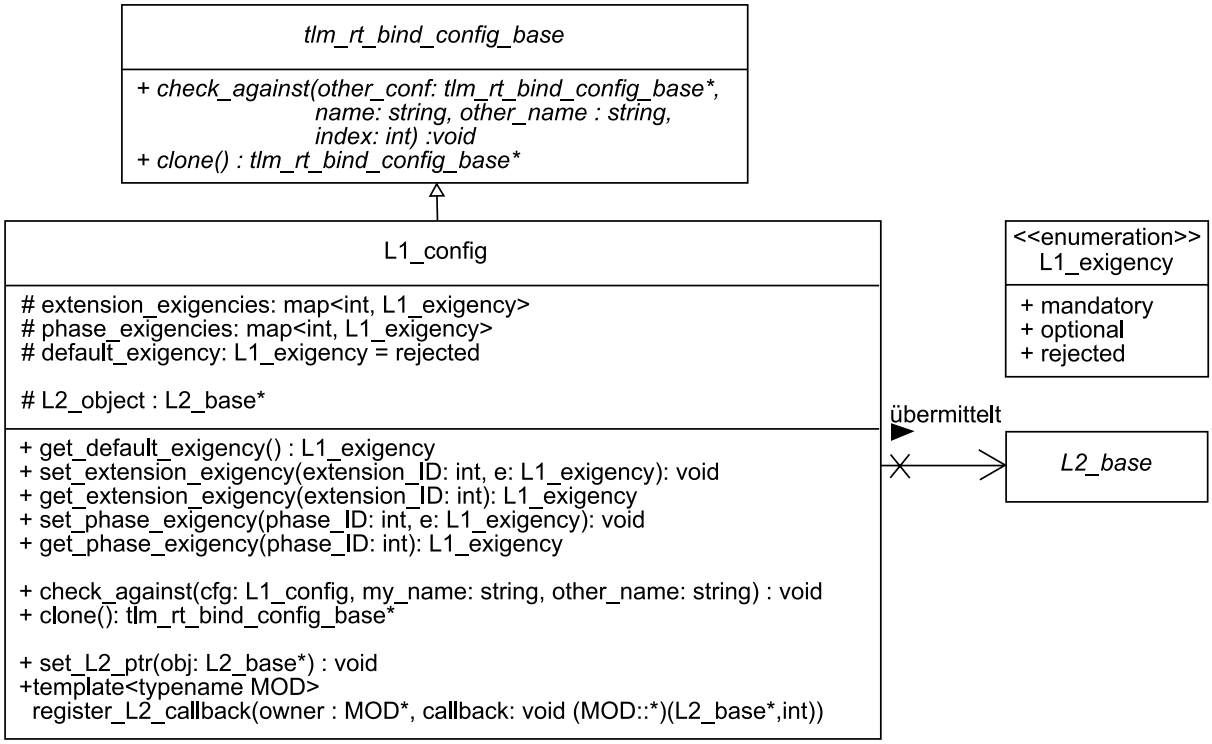


Abbildung 4.24.: Klassendiagramm für die L1-Interoperabilitätskonfiguration

Default-Erfordernisgrad vorhanden, der sich auf alle nicht explizit gesetzten GP-Erweiterungen und TLM-Phasen bezieht.

Funktion `check_against` führt den L1-Test durch (siehe unten) und die Funktionen `set_L2_ptr` und `register_L2_callback` ermöglichen die L2-Tests (siehe unten).

L1- und L2- Interoperabilitätstest

Um L1-Tests durchzuführen, erzeugt der Besitzer eines Sockets eine L1-Konfiguration und übergibt mittels `set_rt_bind_config_ptr` den Zeiger auf diese Konfiguration an den Socket. Anschliessend können die Erfordernisgrade der GP-Erweiterungen und TLM-Phasen gesetzt werden. Möchte der Nutzer des Sockets L2-Tests auf verbundenen Sockets durchführen, so registriert er mit `register_L2_callback` einen Callback auf der L1-Konfiguration. Will er darüber hinaus den verbundenen Sockets L2-Tests ermöglichen, so setzt er mit `set_L2_ptr` einen Zeiger auf ein von `L2_base` abgeleitetes Objekt in der L1-Konfiguration.

Der grundlegende Ablauf der L1- und L2-Tests findet in der Funktion `check_against` der L1-Konfiguration statt und ist in Abbildung 4.25 auf der nächsten Seite dargestellt. `check_against` wird dabei auf einer Kopie der an den Socket übergebenen L1-Konfiguration aufgerufen (siehe Abbildung 4.20). Nach dem Aufruf handelt es sich dann um die resultierende Konfiguration.

Zur Vereinfachung von Nutzer-Code kann durch den `get_config`-IMC ein Null-Zeiger zurückgeliefert werden. Dies bedeutet dann, dass der verhandelbare Satz von GP-Erweiterungen und TLM-Phasen komplett abgelehnt wird. Dementsprechend wird in Abbildung 4.25 zuerst

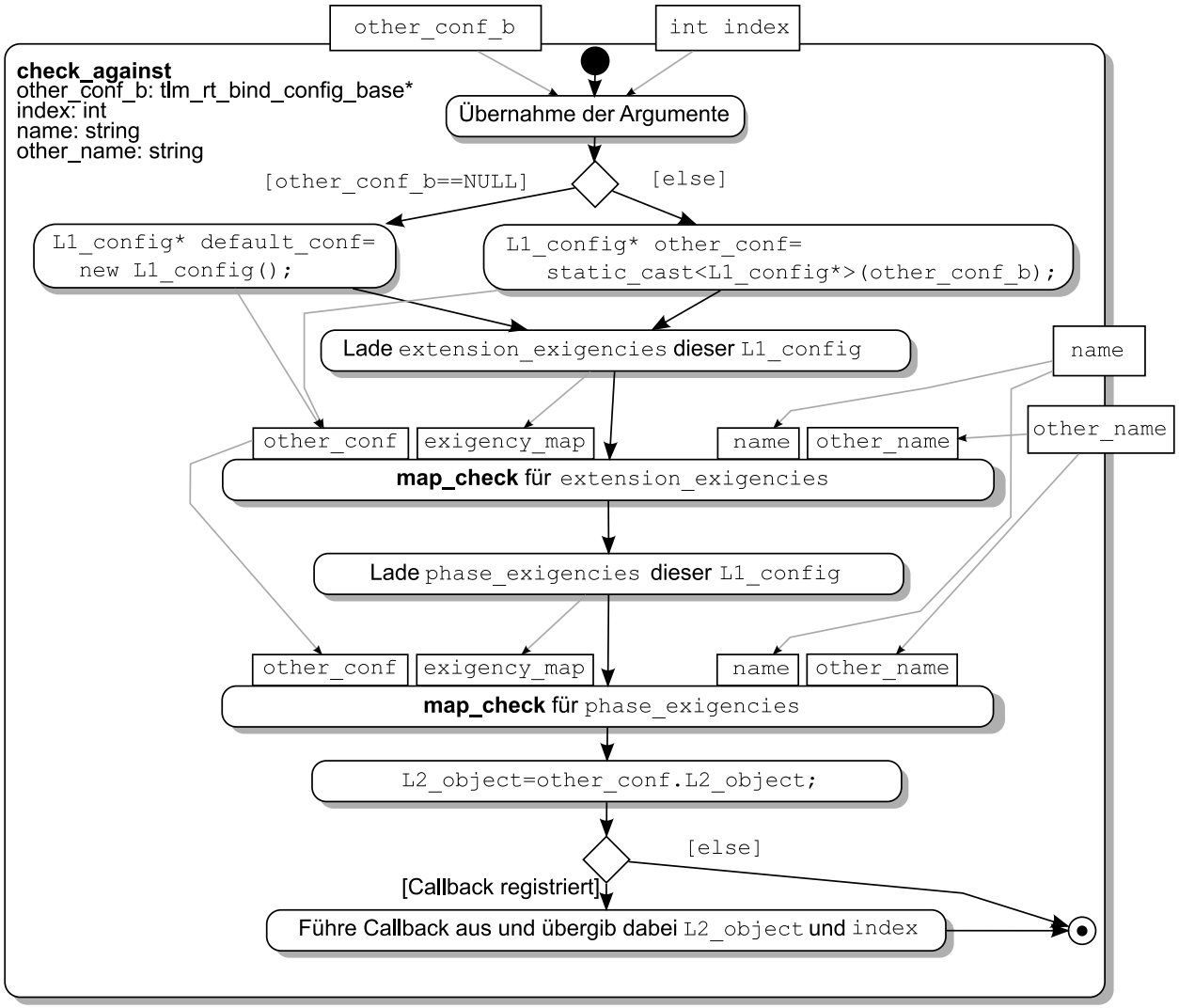


Abbildung 4.25.: Ablauf der L1- und L2-Tests in der `check_against`-Funktion

überprüft, ob der Zeiger auf die Konfiguration des verbundenen Sockets der Null-Zeiger ist. Ist dem so, wird eine leere L1-Konfiguration erstellt, ansonsten wird der übergebene Zeiger in den erwarteten L1-Konfigurationstyp umgewandelt[23]. Anschliessend werden die Map der GP-Erweiterungen und dann die Map der TLM-Phasen überprüft. Der Ablauf dazu ist in Abbildung 4.26 auf der nächsten Seite gezeigt. Danach wird der Zeiger auf die `L2_base` aus der übergebenen Konfiguration in die Konfiguration übernommen, auf der `check_against` aufgerufen wurde. Da `check_against` auf der resultierenden Konfiguration aufgerufen wird, bedeutet dies, dass eine resultierende Konfiguration stets einen Zeiger auf das L2-Objekt des verbundenen Sockets hat. Wurde ein L2-Callback an der an den Socket übergebenen L1-Konfiguration registriert, so ist dieser auch in der Kopie enthalten, auf der `check_against` aufgerufen wird, und wird dann durchgeführt.

Beim Vergleich der Maps der Erfordernisgrade (Abbildung 4.26) werden alle explizit gesetzten Erfordernisgrade untersucht. Wird eine Konstellation gefunden, bei der ein zwingender

[23]Dies kann über einen statischen Cast geschehen, da die TC sicher stellt, dass zwei verbundene Sockets die gleiche Konfigurationsklasse nutzen.

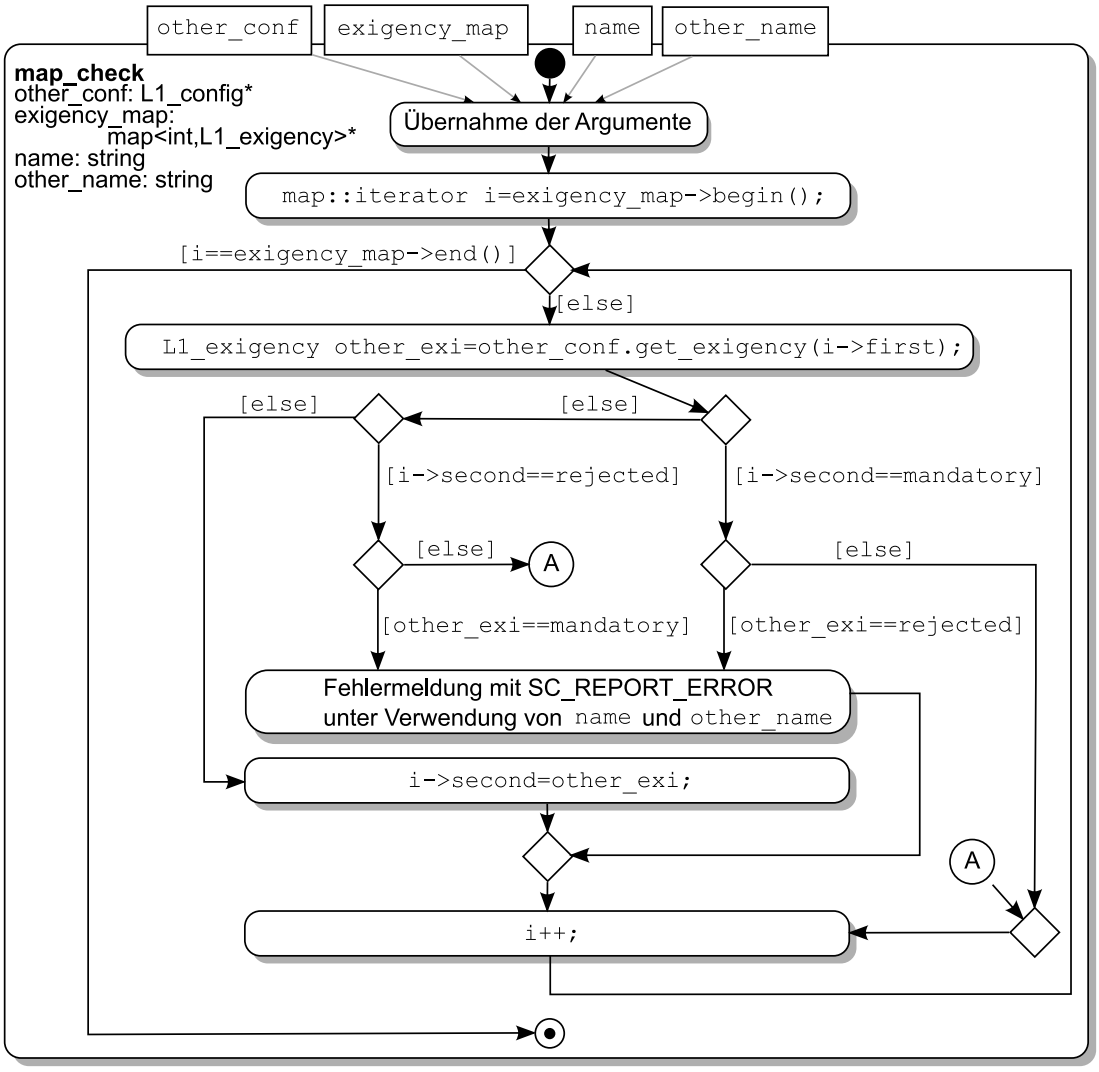


Abbildung 4.26.: Überprüfung der Erfordernisgrade im L1-Test (Die in der zweiten Aktivität verwendete Funktion `get_exigency` repräsentiert die Funktion `get_extension_exigency` bzw. `get_phase_exigency` der `L1_config`, abhängig davon, ob die Map der GP-Erweiterungen oder TLM-Phasen überprüft wird.)

Erfordernisgrad mit einem verbotenen Erfordernisgrad kollidiert, wird ein entsprechender Fehler mit `SC_REPORT_ERROR` ausgegeben. Ist ein Erfordernisgrad auf optional gesetzt, so wird der Erfordernisgrad aus der anderen Konfiguration übernommen (vergleiche Tabelle 4.14). Ein Code-Beispiel zur L1- und L2-Interoperabilitätsprüfung kann Anhang F entnommen werden.

4.7. Klassifikation von Modifiabilities

Wie in Abschnitt 4.5.2 erläutert, ist eine präzise Klassifikation von Modifiabilities sowohl für die GP-Grundelemente als auch für GP-Erweiterungen notwendig. Die von mir vorgeschla-

gene Klassifikation eines GP-Grundelements oder einer GP-Erweiterung bestimmt sich aus zwei Grundeigentschaften: Der Phasenassoziation und dem Änderungsinterval. Im Folgenden werden diese Eigenschaften erläutert.

4.7.1. Phasenassoziation

Wie bereits in Abschnitt 4.5.2 erläutert, ist es im Rahmen der taktgenauen Modellierung notwendig, die TLM-Phasen zu benennen, bei denen ein GP-Grundelement oder eine GP-Erweiterung gültig sein kann. Den Satz von TLM-Phasen, bei denen ein GP-Grundelement oder eine GP-Erweiterung gültig sein kann, nenne ich **Phasenassoziation** eines GP-Grundelements oder einer GP-Erweiterung.

Wie im TLM-Phase-Mapping (siehe Abschnitt 4.3.2) erklärt, wird die Gültigkeit von GP-Grundelementen grundsätzlich durch die TLM-Phase bestimmt. Bei Empfang eines Aufrufes von `nb_transport` mit einer bestimmten TLM-Phase müssen alle GP-Elemente gültig sein, auf die ein Signal der definierenden Busphase der TLM-Phase abgebildet wird. Die Phasenassoziation eines GP-Grundelements bestimmt sich somit wie folgt:

1. Führe GP-Mapping durch.
2. Führe TLM-Phase-Mapping durch[24].
3. Untersuche die definierende Busphase Φ aller TLM-Phasen, deren Namen mit `BEGIN_` oder `ABORT_` beginnen.

3.1 Für jeden Port $p \in PP_{\Phi S}$, der im GP-Mapping auf ein GP-Grundelement abgebildet, füge die aktuell untersuchte TLM-Phase zur Phasenassoziation dieses GP-Grundelements hinzu.

4. Untersuche die definierende Busphase Φ aller TLM-Phasen, deren Namen mit `END_` beginnen.

4.1 Für jeden Port $p \in PP_{\Phi E}$, der im GP-Mapping auf ein GP-Grundelement abgebildet, füge die aktuell untersuchte TLM-Phase zur Phasenassoziation dieses GP-Grundelements hinzu.

Im Gegensatz zu den GP-Grundelementen erzwingt die TLM-Phase nicht die Gültigkeit von GP-Erweiterungen (vgl. Abschnitt 4.3.2). Busphasen, die auf die gleiche TLM-Phase abgebildet werden, können sich in den Signalen unterscheiden, die auf GP-Erweiterungen abgebildet werden. Somit können GP-Erweiterungen bei bestimmten TLM-Phasen gültig sein, müssen sie aber nicht. Die TLM-Phasen, bei denen eine Erweiterung gültig sein kann, ergibt sich nach folgenden Ablauf:

1. Führe GP-Mapping durch.

[24] Eine TLM-Phasen-Reduzierung (siehe Abschnitt 4.3.4) darf noch nicht durchgeführt werden.

2. Führe TLM-Phase-Mapping durch[24].

3. Untersuche jeden Port p jeder Busphase Φ der Peripherie Φ_{Per}

3.1 Gibt es keine GP-Erweiterung auf die Port p im GP-Mapping von Φ abgebildet wurde, brich die Untersuchung von p ab.

3.2 Gehört der Port p zum Start, Abbruch und Ende der Busphase ($p \in PP_{\Phi S} \cap PP_{\Phi E}$), finde die TLM-Phasen, die Start, Ende und Abbruch der Busphase repräsentieren. Füge diese Phasen zur Phasenassoziation der GP-Erweiterung hinzu, auf die Port p im GP-Mapping abgebildet wurde. Brich die Untersuchung von p ab.

3.3 Gehört der Port p nur zum Start und Abbruch der Busphase ($p \in PP_{\Phi S} \backslash PP_{\Phi E}$), finde die TLM-Phasen, die Start und Abbruch der Busphase repräsentieren. Füge diese Phasen zur Phasenassoziation der GP-Erweiterung hinzu, auf die Port p im GP-Mapping abgebildet wurde. Brich die Untersuchung von p ab.

3.4 Gehört der Port p nur zum Ende der Busphase ($p \in PP_{\Phi E} \backslash PP_{\Phi S}$), finde die TLM-Phasen, die das Ende der Busphase repräsentiert. Füge diese Phase zur Phasenassoziation der GP-Erweiterung hinzu, auf die Port p im GP-Mapping abgebildet wurde.

Danach hat jede GP-Erweiterung eine Liste der assoziierten TLM-Phasen. Empfängt ein Modul eine dieser TLM-Phasen, so kann die GP-Erweiterungen gültig sein und muss daraufhin überprüft werden. Ein Mechanismus dafür wird in 4.8.4 vorgestellt.

Die Phasenassoziation von GP-Elementen und GP-Erweiterungen kann im Rahmen der L2-Interoperabilitätstests (oder im Rahmen anderer geeigneter Mechanismen) zwischen zwei verbundenen Sockets eingeschränkt werden. Gibt es solch einen Mechanismus nicht, so sind die mit Hilfe der obene genannten Algorithmen bestimmten Phasenassoziationen verbindlich.

Darüber hinaus ist es zulässig, die Phasenassoziation in Abhängigkeit vom Zustand des GP, also den Werten der GP-Elemente, einzuschränken. Dies ist notwendig, wenn ein GP-Element bei verschiedenen Transaktionen in unterschiedlichen Phasen verwendet wird, jedoch nie in allen. Als Beispiel soll das Datenfeld des GP im OPB-Protokoll dienen: Nach dem GP- und TLM-Phase-Mapping umfasst die Phasenassoziation sowohl `BEGIN_XFER` als auch `END_XFER` (vlg. Anhang D). Das Datenfeld ist jedoch nur bei schreibenden Transaktionen mit `BEGIN_XFER` assoziiert. Die Assoziation zu `END_XFER` besteht nur bei lesenden Transaktionen. Dementsprechend muss hier die Phasenassoziation mit Hilfe des Zustand des Command-Feldes des GP eingeschränkt werden.

4.7.2. Änderungsintervall

Die Untersuchungen zu den Änderungsintervallen habe ich ursprünglich im Rahmen des GreenBus-Projekts [KGB+06],[Green07]$_i$ für die OCP-SLD-WG durchgeführt [Günz07]. Die Ergebnisse wurden als Teil von [BAGK07] veröffentlicht. Dieser Abschnitt fasst die Ergebnisse zusammen und bildet sie auf TLM-2.0 ab.

Ein Wert im GP (ein GP-Grundelement oder eine GP-Erweiterung) wird stets von einem Target oder einem Initiator festgelegt. Interconnects können Werte lediglich ändern, wobei das Hinzufügen einer GP-Erweiterung als Änderung des Wertes der GP-Erweiterung angesehen wird (denn auch die Abwesenheit der GP-Erweiterung repräsentiert einen Wert. Siehe Abschnitt 4.8.2). Ein Wert muss festgelegt werden, bevor erstmals eine Phase aus der Phasenassoziation des Wertes im Verlaufe einer Transaktion übertragen wird[25]. Jeder weitere schreibende Zugriff auf einen solchen Wert ist als Änderung des Wertes zu verstehen.

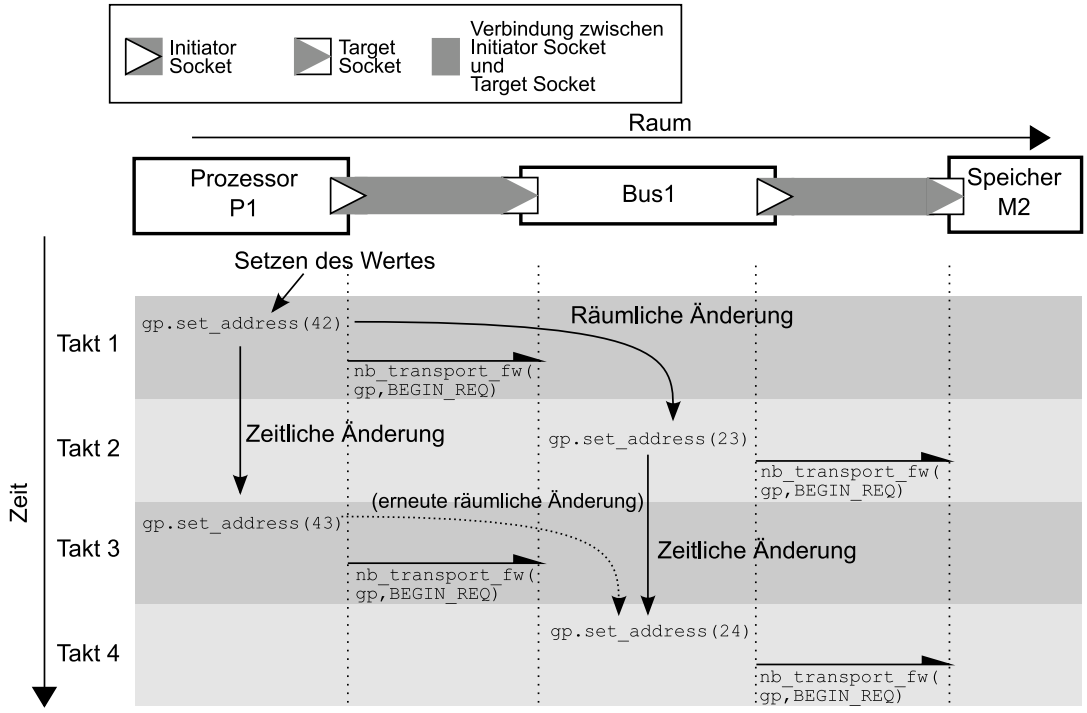


Abbildung 4.27.: Zeitliche und räumliche Änderung der TLM-Phase bei einer Transaktion

Ein Wert kann sich ändern, wenn sich die Transaktion im System ausbreitet, wenn sie also von einer Punkt-zu-Punkt-Verbindung auf die nächste weitergereicht wird. Ein solches Weiterreichen ist nur bei Interconnects der Fall. Eine **räumliche Änderung** eines Wertes bezeichnet somit einen schreibenden Zugriff von einem Interconnect auf einen Wert des GPs, bevor das GP mit `nb_transport` und einer TLM-Phase aus der Phasenassoziation des Wertes übertragen wird. Eine Koordinate in diesem Raum ist dann eine Punkt-zu-Punkt-Verbindung aus dem Transaktionspfad.

Ein Wert kann sich auch ändern, wenn das GP mehrfach in unterschiedlichen Takten mit Phasen aus der Phasenassoziation des Wertes auf der gleichen Punkt-zu-Punkt-Verbindung übertragen wird. Der Wert ändert sich also auf der gleichen Raumkoordinate in Abhängigkeit von der Zeit (in Takten). Eine **zeitliche Änderung** eines Wertes bezeichnet also

[25]Ist es im modellierten MMBIF einem Modul nicht möglich festzustellen, ob im Rahmen der aktuellen Transaktion bereits eine Phase aus der Phasenassoziation auf einer anderen Punkt-zu-Punkt-Verbindung übermittelt wurde, muss es davon ausgehen, dass dies der Fall war und sich so verhalten, als würde es den Wert ändern wollen.

einen schreibenden Zugriff von einem Initiator, Target oder Interconnect auf einen Wert des GPs, bevor das GP erneut im Verlauf der Transaktion mit `nb_transport` und einer TLM-Phase aus der Phasenassoziation des Wertes auf einer Punkt-zu-Punkt-Verbindung übertragen wird.

Abbildung 4.27 verdeutlicht dies anhand einer vereinfachten Darstellung des Beginns eines Multi-Request-Bursts. Man erkennt das Setzen der Adresse in Takt 1 durch den Prozessor. In Takt 2 findet eine räumliche Änderung der Adresse statt, wenn das GP von der linken auf die rechte Punkt-zu-Punkt-Verbindung des Interconnects Bus1 weitergeleitet wird. Dabei ist die Adresse eingangseitig 42 und ausgangsseitig 23. In Takt 3 ist eine zeitliche Änderung der Adresse durch den Initiator auf 43 dargestellt. In Takt 4 gibt es schließlich eine zeitliche Änderung der Adresse durch das Interconnect auf 24, dabei ist dies gleichzeitig auch eine räumliche Änderung.

Tabelle 4.28 listet die möglichen Kombinationen aus räumlichen und zeitlichen Änderungen auf und ordnet ihnen von mir gewählte Bezeichnungen der entsprechenden Änderungsintervalle und deren Kurzzeichen zu. Die folgenden Abschnitte erläutern die einzelnen Änderungsintervalle und begründet, warum bei zeitlicher Varianz die räumliche Varianz nicht mehr relevant ist (bezüglich der Zuordnung des Änderungsintervalls).

Änderung		Änderungsintervall	Kurzzeichen
Räumlich	Zeitlich		
konstant	konstant	Ende-zu-Ende-invariant	e2e
variabel	konstant	Punkt-zu-Punkt-invariant	x2x
konstant	variabel	Punkt-zu-Punkt-variabel	p2p
variabel	variabel		

Tabelle 4.28.: Änderungsintervalle bei der Transaction-Level-Modellierung

4.7.3. Ende-zu-Ende-Invarianz

Da Ende-zu-Ende-invariante Werte sich weder räumlich noch zeitlich ändern dürfen, können sie nur einmal von Initators oder Targets gesetzt werden und bleiben von diesem Zeitpunkt an bis zum Ende der Transaktion konstant. Aufgrund dieser Konstanz des Wertes ist ein Auslesen aus dem GP nach Empfang bzw. nach dem Senden einer Phase aus der Phasenassoziation bis hin zum Ende der Transaktion möglich.

Der Term „Ende-zu-Ende“ bezieht sich darauf, dass der Wert zwischen den Endpunkten der Transaktion, also zwischen Initiator und Target, nach der Ausbreitung einer Phase aus der Phasenassoziation immer gleich ist.

Sollte ein Interconnect zur korrekten Modellierung seines Verhaltens einen solchen Wert setzen bzw. ändern müssen, so muss das Interconnect eine Kopie der Transaktion anlegen,

die Änderung in der Kopie vornehmen und die Kopie weitersenden. Dadurch wird das Interconnect zum Target für den ursprünglichen Initiator und zum Initiator für das finale Target und hält somit alle Regeln ein.

	Setzen	Lesen
Wer	Initiator oder Target	Initiator, Target und Interconnects
Wann	Einmalig vor erstem Senden des GP mit einer Phase aus der Phasenassoziation	Nach dem ersten Empfang des GP mit einer Phase aus der Phasenassoziation bis zum Ende der Transaktion

Tabelle 4.29.: Regeln für Ende-zu-Ende-invariante Werte

4.7.4. Punkt-zu-Punkt-Invarianz

Da sich Punkt-zu-Punkt-invariante Werte räumlich ändern dürfen, können Interconnects diese einmalig bei der ersten Weiterleitung einer Phase aus der Phasenassoziation ändern. Weitere Änderungen sind nicht zulässig, da es sich bei diesen stets auch um zeitliche Änderungen handeln würde.

Die zeitliche Invarianz ist aufgrund der Shared-Data-Problematik (siehe Abschnitt E.3) aber lediglich eine gewünschte Invarianz, beobachtet man den Wert auf einer Punkt-zu-Punkt-Verbindung, ist auch eine zeitliche Varianz zu beobachten. Aus diesem Grund muss ein Modul, das einen x2x-Wert liest oder setzt und diesen später im Verlauf der Transaktion noch einmal benötigt, eine Kopie des Wertes anlegen und geeignet speichern. Abbildung 4.30 auf der nächsten Seite illustriert die Abweichung zwischen gewünschter Invarianz und beobachtbarem Wert aufgrund der Shared-Data-Problematik anhand der Adresse eines GP: Ein Initiator sendet in Takt 1 ein GP mit Adresse 42 an ein Interconnect. Danach darf der Initiator die Adresse nicht mehr ändern, aus seiner Sicht bleibt sie also bei 42. In Takt 2 ändert das Interconnect die Adresse auf 23 und sendet das GP an das Target. Dementsprechend ist die effektive Adresse auf der Verbindung zwischen Interconnect und Target 23. In Takt 3 sendet nun der Initiator das GP erneut (mit einer neuen Phase) zum Interconnect. Würde das Interconnect jetzt die Adresse aus dem GP lesen, so würde es eine 23 sehen, da die Änderung der Adresse in Takt 2 wegen der Shared-Data-Problematik nicht auf eine Punkt-zu-Punkt-Verbindung beschränkt ist. Da die Adresse im Beispiel aber ein x2x-Änderungsinterval besitzt, weiß das Interconnect, dass der Wert der Adresse nicht mehr vertrauenswürdig ist. Es behandelt das GP so, als wäre die Adresse nach wie vor 42. Dazu hat das Interconnect in Takt 1 oder 2 nach Empfang des GP, aber vor Änderung der Adresse in geeigneter Art und Weise eine Kopie der Adressinformation gespeichert[26].

[26]Dieses Speichern erfolgt entweder über die Verwendung von sog. Instanz-spezifischen Erweiterungen (siehe TLM-2.0-LRM) oder aber erfolgt automatisch im internen Zustand der Module, wie z.B. als Zustand des Addressdekoders. Der Aufwand dafür ist i.d.R. sehr gering.

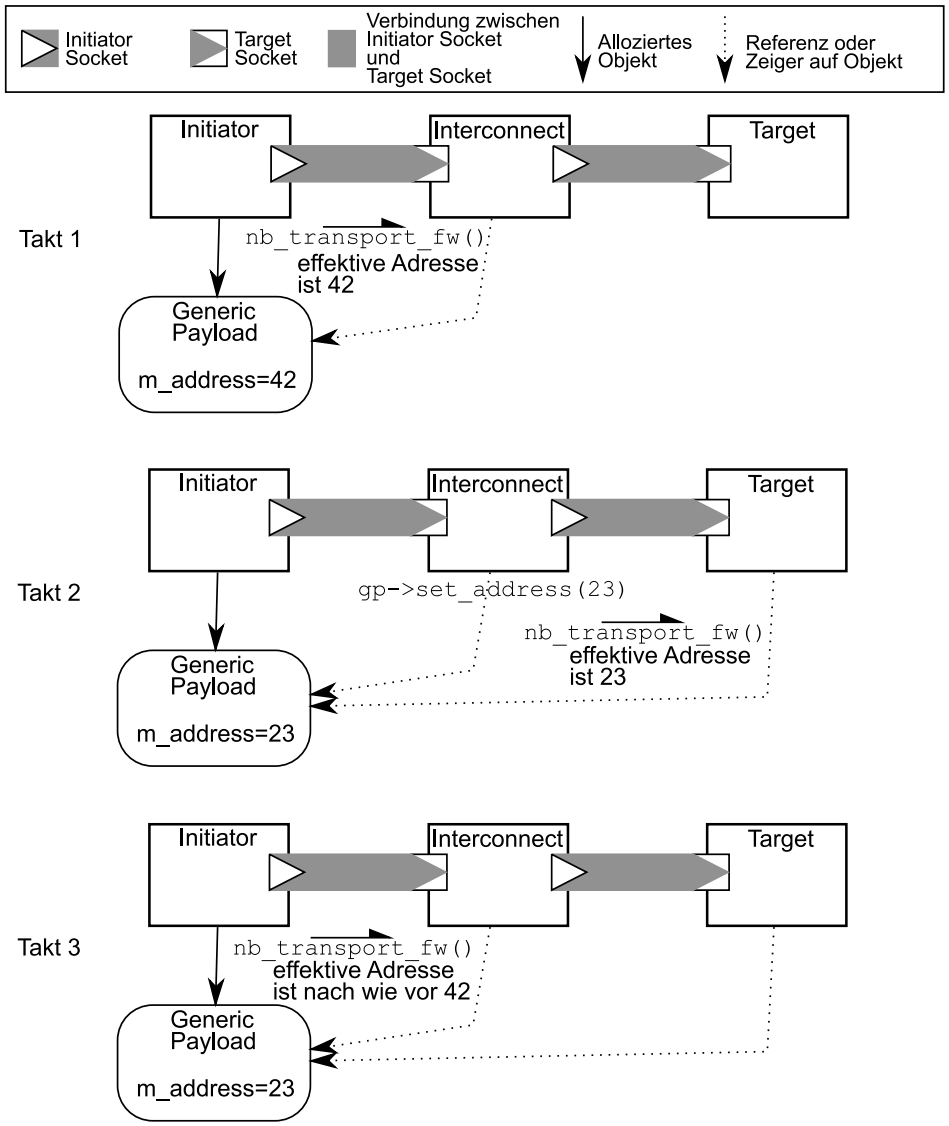


Abbildung 4.30.: Effektive zeitliche Invarianz bei beobachtbarer zeitlicher Varianz

An dieser Stelle ist es wichtig, auf einen Sonderfall hinzuweisen. Kann ein Interconnect I aus seiner Sicht der erste Sender einer Phase aus der Phasenassoziation eines x2x-Wertes sein, so legt es vor dem Senden den Wert fest. Ein Empfänger E (z.B. ein Target) geht dann davon aus, dass der Wert vom ersten Empfang bis zur ersten Weiterleitung des GP mit einer TLM-Phase aus der Phasenassoziation vertrauenswürdig bleibt. Jedoch könnte in dieser Zeit ein anderes Modul (z.B. der Initiator) erstmals eine Phase aus der Phasenassoziation des Wertes an das Interconnect I senden und den Wert dementsprechend setzen. Aufgrund der Shared-Data-Problematik bleibt also aus Sicht des Empfängers E der Wert zwischen Empfang und Weiterleitung nicht mehr stabil. In einem solchen Fall muss ein x2x-Wert, obwohl er lediglich räumlich veränderbar ist, wie ein p2p-Wert (siehe unten) behandelt werden. Das bedeutet, dass ein solcher Wert tatsächlich ein x2x-Änderungsintervall besitzt, aber die Regeln für p2p-änderbare Werte Anwendung finden müssen. Um dies für den Nutzer nicht zu kompliziert zu machen, sollten solche x2x-Werte als p2p-änderbar festgelegt werden.

	Setzen	Lesen
Wer	Initiator, Target und Interconnects	Initiator, Target und Interconnects
Wann	Einmalig vor erstem Senden des GP mit einer Phase aus der Phasenassoziation	Nach dem ersten Empfang des GP mit einer Phase aus der Phasenassoziation bis zum erstem Senden des GP mit einer Phase aus der Phasenassoziation

Tabelle 4.31.: Regeln für Punkt-zu-Punkt-invariante Werte

4.7.5. Punkt-zu-Punkt-Varianz

Punkt-zu-Punkt-Varianz deckt Werte ab, die zeitlich variabel sind, unabhängig von ihrer räumlichen Varianz. Um dies zu erläutern, betrachte man vorerst Werte, die sowohl zeitlich als auch räumlich variabel sind. Diese Werte dürfen von jedem Modul auf dem Transaktionspfad vor jedem `nb_transport`-Aufruf mit einer Phase aus der Phasenassoziation gesetzt werden. Als Folge aus der Shared-Data-Problematik ergibt sich aber, dass die Werte vor jedem `nb_transport`-Aufruf mit einer Phase aus der Phasenassoziation gesetzt werden müssen. Abbildung 4.32 belegt dies. Dargestellt ist ein Initiator, der über ein Interconnect mit einem Target verbunden ist. Der Initiator sendet hintereinander zweimal die Phase `BEGIN_REQ` im Verlauf einer Transaktion zum Interconnect. Das Interconnect sendet diese Phase einen Takt verzögert an das Target weiter. Im GP existiert eine GP-Erweiterung namens `req_tag`, die eine im Beispiel nicht näher bestimmte Information trägt, die sich mit jeder Phase ändern kann und darüber hinaus auch noch vom Interconnect verändert werden dürfte. Es handelt sich also um eine Punkt-zu-Punkt-variable GP-Erweiterung. In Abbildung 4.32 sind zwei Transaktionen dargestellt. Bei der ersten muss das Interconnect den Wert ändern, bei der zweiten nicht.

Interessant ist zu beobachten, welche Werte das Target empfängt. In der ersten Transaktion wird bei Empfang von `nb_transport_fw` das Target immer den Wert aus dem GP lesen, den das Interconnect zu senden beabsichtigte, da der Wert unmittelbar vor dem Aufruf gesetzt wurde. In der zweiten Transaktion ist dies nicht immer gegeben. Wird in Takt 6 erst der Simulationsprozess im Interconnect ausgeführt, dann wird das GP mit `req_tag=75`, also wie beabsichtigt, gesendet. Wird hingegen in Takt 6 erst der Simulationsprozess im Initiator ausgeführt (dieser Fall ist in Abbildung 4.32 dargestellt), so setzt er `req_tag=76`. Wird dann der Simulationsprozess im Interconnect ausgeführt, liest das Target genau diesen Wert aus dem GP (Shared-Data-Problematik), obwohl es `req_tag=75` hätte lesen müssen. Um dies zu vermeiden, muss vor dem Aufruf von `nb_transport` jeder Punkt-zu-Punkt-variable Wert richtig gesetzt werden, selbst wenn dies der Wert bei Empfang des GP war. Als Folgerung daraus ergibt sich auch, dass bei Empfang von `nb_transport` jeder Punkt-zu-Punkt-variable

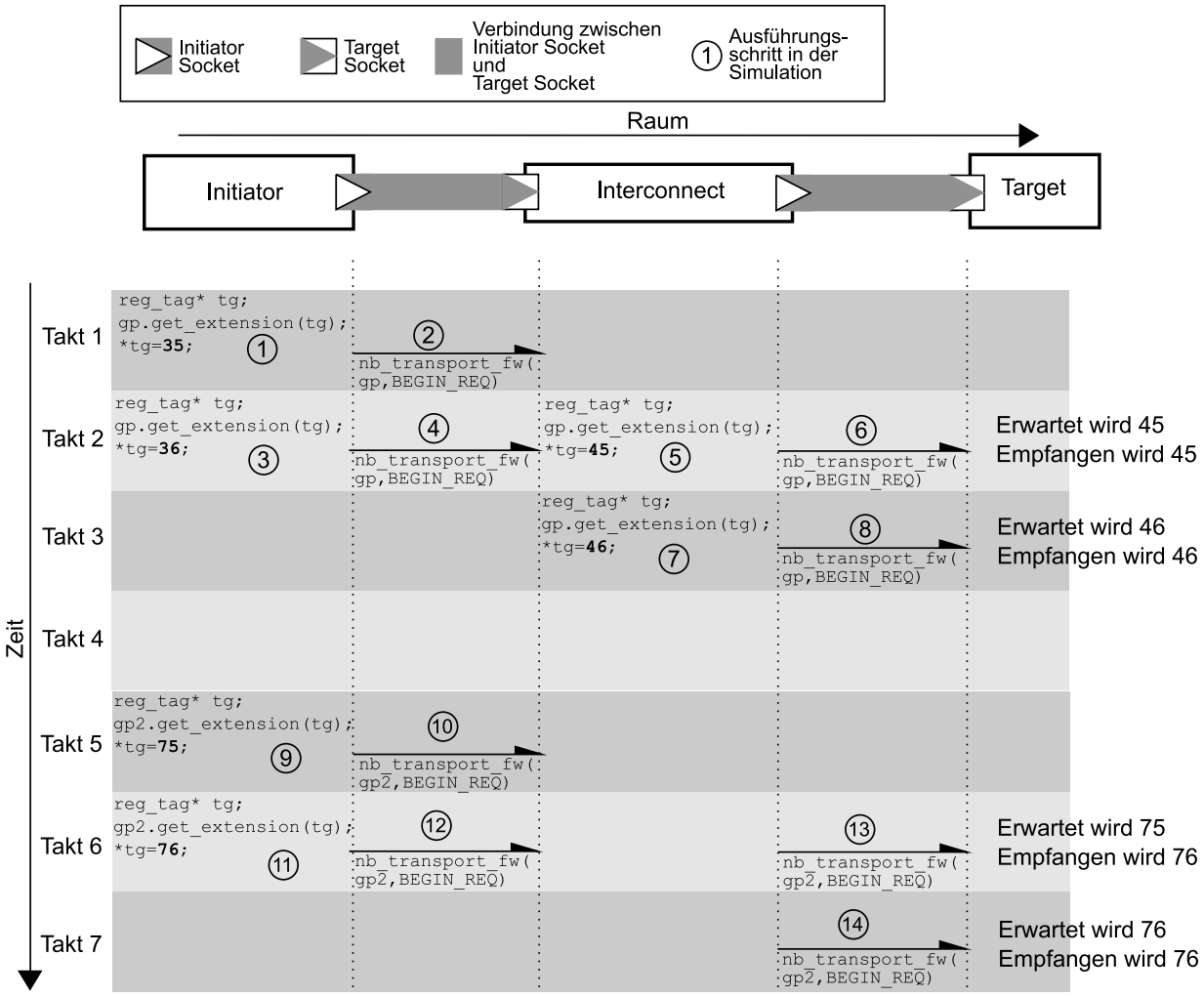


Abbildung 4.32.: Mögliche Fehlsimulation, wenn p2p-Werte nicht gesetzt werden.

Wert ausgelesen und vor jedem Senden wieder gesetzt werden muss, da er sich potentiell in der Zeit zwischen Empfang und Weiterleitung ändern kann[27].

Man betrachte nun Werte, die zeitlich variabel, aber räumlich konstant sind, deren Wert also von Interconnects nicht geändert werden darf. Dies ist genau in der zweiten Transaktion in Abbildung 4.32 der Fall. Wie oben bereits erläutert wurde, muss das Interconnect aber, damit das Target die richtigen Werte empfängt, vor jedem `nb_transport`-Aufruf die Werte setzen. Das heißt, gerade weil das Interconnect die Werte nicht ändern darf, muss es sie vorm Senden immer auf die Empfangenen setzen. Somit verhält sich ein Interconnect bei zeitlich variablen Werten, unabhängig von der räumlichen Varianz, immer gleich: Vor einem `nb_transport`-Aufruf muss der Wert gesetzt werden. Da sich die räumliche Varianz nur auf Interconnects bezieht, sind Werte mit zeitlicher Varianz auch aus Sicht von Initiators und Targets unabhängig von der räumlichen Varianz immer gleich zu behandeln. Der eigentliche Unterschied liegt darin, dass bei räumlicher Konstanz das Interconnect den Wert setzen

[27] Einzige Ausnahme ist ein direktes Weiterleiten des Aufrufes ohne Kontextwechsel zwischen Empfang und Senden.

muss, den es empfing, während bei räumlicher Varianz der Wert geändert werden darf. Da sich aber die Regeln nicht auf die Frage beziehen, welcher Wert gesetzt wird, sondern auf die Frage, ob der Wert gesetzt wird, muss für die Festlegung des Änderungsintervalls nicht zwischen beiden Fällen unterschieden werden.

	Setzen	Lesen
Wer	Initiator, Target und Interconnects	Initiator, Target und Interconnects
Wann	Immer vor dem Senden des GP mit einer Phase aus der Phasenassoziation	Nur innerhalb des IMC beim Empfang des GP mit einer Phase aus der Phasenassoziation

Tabelle 4.33.: Regeln für Punkt-zu-Punkt-variante Werte

4.7.6. Diskussion der Änderungsintervalle

Die Phasenassoziation eines GP-Grundelements oder einer GP-Erweiterung kann wie in Abschnitt 4.7.1 erläutert ermittelt werden. Für die Bestimmung des Änderungsintervalls gibt es keinen solchen Algorithmus. Das Änderungsintervall muss bei der Festlegung der Modellierung eines bestimmten Protokolls festgelegt werden. Anschließend legt die Types-Class (implizit, über Regeln) neben den Sätzen der unverhandelbaren und verhandelbaren GP-Erweiterungen und TLM-Phasen auch die Änderungsintervalle der GP-Grundelemente und GP-Erweiterungen (im weiteren Verlauf dieses Abschnittes wird zusammenfassend nur noch von GP-Elementen gesprochen) fest. Das heißt für ein gegebenes Protokoll sind die Änderungsintervalle fix. Dieser Abschnitt diskutiert, welche Überlegungen angestellt werden müssen, wenn für ein Protokoll die Änderungsintervalle festgelegt werden.

Wie oben beschrieben sind GP-Elemente mit e2e-Änderungsintervall aus Nutzersicht am einfachsten zu handhaben, da sie, nachdem die erste Phase aus der Phasenassoziation gesendet wurde, bis zum Ende der Transaktion konstant bleiben und so stets einfach über das GP ausgelesen werden können. Aufgrund ihrer räumlichen Veränderbarkeit sind GP-Elemente mit x2x-Änderungsintervall nur beim ersten Empfang einer Phase aus der Phasenassoziation gültig. Benötigt ein Modul im späteren Verlauf der Transaktion noch einmal den Wert, so muss er beim ersten Empfang bzw. ersten Senden geeignet gespeichert werden. Zwischen dem ersten Empfang und der Weiterleitung bleibt der Wert aber garantiert stabil, sodass Kontextwechsel in diesem Zeitraum unkritisch sind. Wie in Abschnitt 4.7.5 beschrieben, müssen p2p-GP-Elemente vor jedem Senden gesetzt und bei jedem Empfang ausgelesen und geeignet gespeichert werden[28]. Darüber hinaus ist der Wert nur innerhalb des `nb_transport`-Aufrufs stabil. Das heißt, nach einem Kontextwechsel kann der Wert nicht mehr als korrekt angesehen werden. Er muss also immer direkt im IMC verarbeitet werden. Aus Sicht des Anwenders

[28]Einzige Ausnahme ist die direkte Weiterleitung eines `nb_transport` Aufrufes.

sind also Ende-zu-Ende-invariante Werte am einfachsten zu handhaben, während Punkt-zu-Punkt-variable Werte am aufwändigsten sind. Punkt-zu-Punkt-invariante Werte liegen im Aufwand ihrer Handhabung zwischen den beiden anderen Änderungsintervallen.

Aus diesem Grund sollte versucht werden, die Anzahl der e2e- und x2x-Elemente zu maximieren, und der p2p-Elemente zu minimieren. Es liegt nahe, e2e-Elemente zu favorisieren, jedoch kann dies, wie unten beschrieben, auch Nachteile haben.

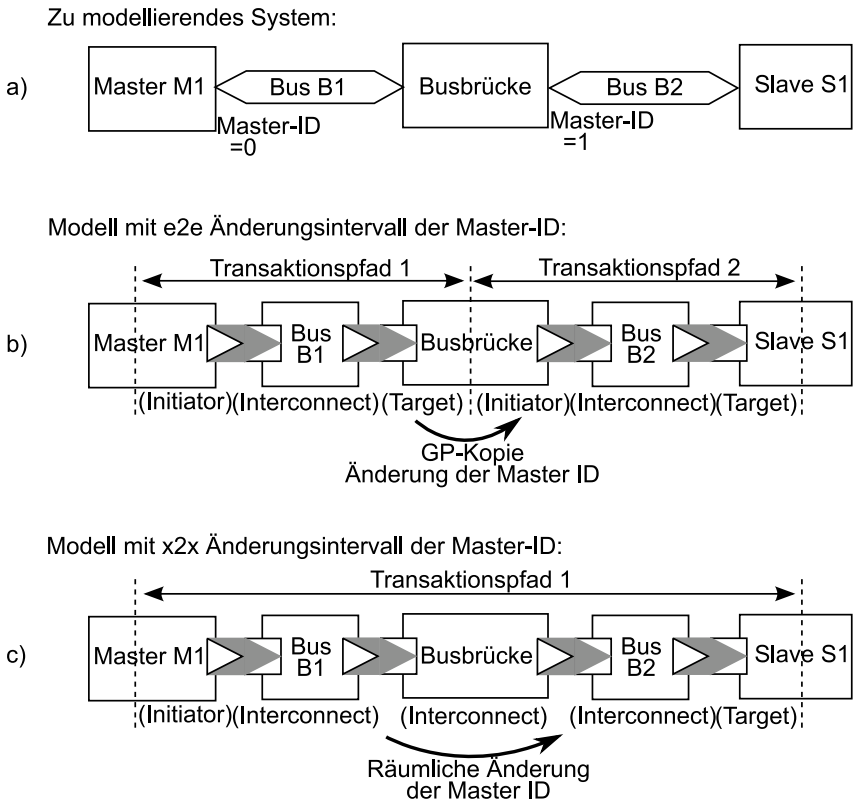


Abbildung 4.34.: Möglicher negativer Seiteneffekt des e2e-Änderungsintervalls

Man betrachte Abbildung 4.34. Dort sind das reale System (Abbildung 4.34a) und dessen TLM-Modell abgebildet (Abbildung 4.34b bzw. c). Ein Prozessor ist über zwei Busse mit gleichem MMBIF mit einem Speicher verbunden, aus dem Daten gelesen werden sollen. Nehmen wir an, dass das MMBIF der Busse eine Master-ID kennt, die vom Master als Absenderkennung gesetzt werden muss. Diese ist zeitlich konstant, das Änderungsintervall kann also e2e oder x2x sein. Ist nun das Änderungsintervall der GP-Erweiterung, die die Master-ID trägt, als e2e festgelegt, so ergeben sich die TLM-Rollen (Initiator, Interconnect, Target) wie in Abbildung 4.34b gezeigt. Man erkennt, dass die Busbrücke Target und Initiator und kein Interconnect ist. Dies begründet sich dadurch, dass die Brücke die Master-ID ändern muss. Aufgrund des e2e-Änderungsintervalls ist es der Busbrücke nicht gestattet, eine räumliche Ändeungs vorzunehmen. Daher muss sie eine Kopie des GP anlegen, in der sie als Initiator die Master-ID setzen darf. Eine solche Kopie bedeutet einen signifikanten Simulations-Overhead (kopierende Iteration über Daten-Feld, Byte-Enable-Feld und Feld der GP-Erweiterungszeiger). Die Vorteile in der Handhabung der e2e-veränderbaren Master-ID-GP-Erweiterung werden durch starke Simulationsgeschwindigkeitseinbußen erkauft. Hätte

die Erweiterung ein x2x-Änderungsintervall, könnte die Busbrücke als Interconnect agieren und eine räumliche Änderung direkt im empfangenen GP vornehmen (Abbildung 4.34c).

Die Entscheidung, ob eine GP-Erweiterung oder ein GP-Grundelement ein e2e-Änderungsintervall erhält, muss also dadurch bestimmt werden, wie hoch die Wahrscheinlichkeit ist, dass in einem (vernünftigen) System eine räumliche Änderung bei einem zeitlich konstanten Wert auftritt. Nur wenn diese Wahrscheinlichkeit sehr gering ist, sollte das e2e-Änderungsintervall verwendet werden.

Das p2p-Änderungsintervall muss grundsätzlich vergeben werden, sobald eine zeitliche Veränderung eines Wertes möglich ist. Davon kann unter speziellen Voraussetzungen abgesehen werden: Unter Umständen durchläuft ein Wert eines GP-Elements bei korrekter Verwendung des modellierten MMBIF zeitlich eine von einem Startwert aus berechenbare Folge. Ist der Verwendungszweck des Modells nicht die Verifikation der korrekten Einhaltung des MMBIF, d.h. wird das Protokoll des MMBIF stets respektiert, wie es zum Beispiel im Falle der Performance- oder Power-Analyse der Fall ist, so kann das p2p-Änderungsintervall auf x2x oder gar e2e geändert werden. In diesem Falle übermittelt das GP-Element nur den Startwert der Folge und alle Module, die die zeitlich variablen Werte benötigen, berechnen diese in Abhängigkeit des Protokollfortschritts. Dies ist in der Regel effizienter und einfacher, als das p2p-Änderungsintervall zu nutzen. Ein Beispiel hierfür ist die Adresse beim AMBA AHB: Diese wird zwar mit jedem einzelnen Request eines Bursts übertragen und ist in jedem Request anders, jedoch unterstützt der AHB nur feste Addresssequenzen, das heißt Abläufe von Adressen bei denen unter Kenntnis der Adresse des ersten Requests die Adressen der Folgerequests berechenbar sind. In einem Modell zur Performance-Evaluation, bei der vom protokollkonformen Verhalten aller beteiligten Module ausgegangen werden kann, würde die Adresse als x2x-veränderbar immer nur im ersten Requests übertragen werden.

Eine weitere Möglichkeit, p2p nach x2x oder e2e abzuändern, ist durch die Verwendung von Feldern gegeben. Diese Möglichkeit wird im nächsten Abschnitt erläutert.

4.7.7. Felder von Daten

Beim GP-Mapping werden die verschiedenen Ports grundsätzlich auf einfache C++-Datentypen abgebildet. Die einzigen Ausnahmen bilden die GP-Grundelemente `m_data` und `m_byte_enable`, welches beides Felder sind. Damit können in den abstrakten Modellierungsstilen AT und LT beliebig viele Datenbytes und deshalb auch beliebig viele Byte-Enables übertragen werden. Somit können Burst beliebiger Länge als einzelne Transaktionen übertragen werden.

Bei der taktgenauen busphasenbasierten *J-R*-Simulation mit TLM-2.0 ist prinzipiell kein Feld notwendig, um die Daten und Byte-Enables eines Burst zu modellieren. Daten und Byte-Enables können als einfache GP-Elemente modelliert werden. Da sie in jeder Phase eines Bursts, die Daten übermittelt, einen anderen Wert haben können, sind sie zeitlich variabel und haben somit ein p2p-Änderungsintervall.

Legt man diese GP-Elemente aber als Felder aus (für Daten und Byte-Enables ist dies durch das GP bereits vorgegeben), füllt sich das Daten- und Byte-Enable-Feld mit jeder Phase ein wenig mehr, jedoch ist ein einmal gefülltes Byte des Datenfeldes nicht mehr änderbar. Das bedeutet, dass im Datenfeld jedes Byte mit einem speziellen Auftreten einer Phase aus der Phasenassoziation des Datenfeldes gekoppelt ist. Dies soll an einem Beispiel verdeutlicht werden. Nehmen wir der Einfachheit halber an, die Phasenassoziation des Datenfeldes enthält nur die TLM-Phase `BEGIN_DATA` und wir betrachten gerade eine Punkt-zu-Punkt-Verbindung mit einer Datenbreite von 4 Byte. Dann sind die Bytes 0, 1, 2 und 3 des Datenfeldes mit dem ersten Auftreten von `BEGIN_DATA`, die Bytes 4, 5, 6 und 7 mit dem zweiten Auftreten von `BEGIN_DATA` assoziiert und so weiter. Das Änderungsintervall aller Bytes ist aber e2e. Das heißt, die Bytes 0, 1, 2, 3 müssen vor dem Senden der ersten `BEGIN_DATA`-Phase gesetzt werden und bleiben von da an stabil, während die Bytes 4, 5, 6, 7 vor dem Senden der zweiten `BEGIN_DATA`-Phase gesetzt werden müssen und von da an stabil sind. Das gleiche gilt für das Byte-Enable-Feld. Man beachte, dass diese Byte-weise Phasenassoziation von der Datenbreite einer Punkt-zu-Punkt-Verbindung abhängt. Sie ist also potentiell auf verschiedenen Punkt-zu-Punkt-Verbindungen unterschiedlich.

So können also die eigentlich p2p-veränderlichen Daten und Byte-Enables auf byteweise e2e-veränderliche Datenfelder abgeändert werden. Dieses Vorgehen kann auch für GP-Erweiterungen genutzt werden. Die GP-Erweiterungen enthalten dann kein einzelnes Element mehr, sondern ein ganzes Feld von Elementen. Es mag nun verlockend erscheinen, jedes p2p-veränderbare Element auf ein byteweise e2e- oder x2x-veränderbares Feld abzuändern, jedoch bedeutet die Existenz eines Feldes einen signifikanten Kopieraufwand im Falle einer Transaktionskopie (wenn z.B. nur ein Byte des byteweise e2e-veränderlichen Datenfeldes geändert werden muss). Die Entscheidung für ein Feld sollte getroffen werden, wenn ein Rückgriff auf ältere Signalwerte in der Funktion eines Moduls wahrscheinlich ist, wenn sehr viele Module Zugriff auf verschiedene Werte gleichzeitig brauchen und wenn die Breite des modellierten Signals auf verschiedenen Punkt-zu-Punkt-Verbindungen variiert (wie im Falle des Datenfeldes). Im letzteren Fall ermöglicht die Punkt-zu-Punkt-verbindungsabhängige, byteweise Phasenassoziation und Modifiability, verschiedene Elemente des Feldes auf verschiedenen Verbindungen als gültig zu markieren.

Die byteweise Modifiability eines Feldes bezieht sich nur auf deren Element, offen aber bleibt die Frage, wann ein Feld erzeug wird und von wem. Dafür sollen hier keine allgemeingültigen Regeln formuliert werden, jedoch triff in den meisten Fällen Folgendes zu: Dem Initiator ist bei Erzeugung des GPs die exakte oder aber zumindest maximale Anzahl von Transfers innerhalb der beabsichtigten Transaktion bekannt. Dementsprechend kann er alle Felder auf die entsprechende Größe festlegen. Dies gilt insbesondere für das Daten- und Byte-Enable-Feld, kann aber auch für GP-Erweiterungen mit Feldern angewendet werden.

4.8. GP-Erweiterungsregeln

Die in Abschnitt 4.7 eingeführte Klassifikation von Modifiabilities mit Phasenassoziation und Änderungsintervall erlaubt es, den GP-Grundelementen im Rahmen des Modells eines bestimmten MMBIF eine eindeutige Modifiability zuzordnen, aus der sich dann Zugriffsregeln (wer, was, wann) ergeben.

Diese Zugriffsregeln gelten grundsätzlich auch für GP-Erweiterungen, jedoch werden GP-Erweiterungen im Gegensatz zu den GP-Elementen außerhalb des GPs von diversen Modulen instanziiert (siehe Abschnitt E.3). Somit kommt zusätzlich zu der Problematik der enthaltenen Werte und deren Gültigkeit noch die Problematik der Speicherverwaltung hinzu. Die Erzeugung und Zerstörung der GP-Erweiterungen muss klar geregelt sein, wobei die Regeln neben der sicheren Speicherverwaltung auch die Simulationsperformance und die Komplexität des Anwendercodes nicht außer Acht lassen dürfen.

4.8.1. Speicherverwaltung von GP-Erweiterungen

Wie in Anhang E beschrieben, sind GP-Erweiterungen C++-Objekte, die von einer speziellen Basisklasse abgeleitet sind. Im GP können Zeiger auf solche Objekte abgelegt werden, sodass diese Objekte über das GP erreichbar sind. Jedoch sind die Objekte, im Gegensatz zu den GP-Grundelementen, aus Sicht der Speicherverwaltung nicht unmittelbar an das GP gekoppelt. Wird eine GP-Instanz gelöscht, so führt dies nicht notwendigerweise zum Löschen der Erweiterung. TLM-2.0 bietet daher Möglichkeiten, die Speicherverwaltung von GP-Erweiterungen an die Speicherverwaltung des GP zu koppeln. Dazu müssen an dieser Stelle kurz die entsprechenden Mechanismen skizziert werden; Details dazu können dem TLM-2.0-LRM entnommen werden.

Für das GP existieren zwei Speicherverwaltungsmodi: Ad-hoc-Speicherverwaltung oder von einem Memory-Manager (MM) gesteuerte Speicherverwaltung. Bei der Ad-hoc-Speicherverwaltung wird vom Initiator ein GP erzeugt und anschließend eine Transaktion mit dem GP durchgeführt. Nach dem Abschluss der Transaktion kann der Initiator das GP direkt wieder zerstören oder für eine neue Transaktion wiederverwenden. Diese Art der Speicherverwaltung kann nur verwendet werden, wenn sichergestellt ist, dass die Transaktion auf allen Verbindungen auf dem Transaktionspfad beendet ist, wenn sie aus Sicht des Initiators endet. Dies kann nur bei Verwendung von `b_transport`, `transport_dbg` oder `get_direct_mem_ptr` garantiert werden, ist also gerade nicht für `nb_transport` geeignet. Bei `nb_transport` (also auch im Rahmen dieser Arbeit) muss die MM-Speicherverwaltung genutzt werden.

Bei der MM-Speicherverwaltung wird dem vom Initiator erzeugten GP ein Zeiger auf ein von der Klasse `tlm_mm_interface` abgeleitetes Objekt übergeben. Die einzige Funktion in dieser Basisklasse ist `void free(tlm_generic_payload*)`. Daneben enthält das GP auch einen bei der Ad-hoc-Speicherverwaltung ungenutzten Referenzzähler. Dieser muss bei der MM-Speicherverwaltung vor dem ersten Senden des GP im Rahmen einer Transaktion vom Initiator auf Eins gesetzt werden. Jedes Interconnect und Target, das das GP länger benötigt

als die Transaktion auf dem Targetsocket des jeweiligen Moduls andauert, muss den Zähler um Eins erhöhen und später, wenn es das GP nicht mehr benötigt, wieder um Eins reduzieren. Der Initiator reduziert den Referenzzähler um Eins, wenn das GP aus seiner Sicht nicht mehr benötigt wird. Der Initiator darf das GP dann nicht direkt wieder verwenden und auf keinen Fall löschen. Letztendlich wird der Referenzzähler im GP den Wert Null erreichen, woraufhin das GP auf dem Zeiger auf das `tlm_mm_interface` die Funktion `free` aufruft. Es obliegt nun dem MM-Objekt, das GP zu löschen oder für die Wiederverwendung vorzubereiten. Das Löschen wird den Destruktor des GP aufrufen, die Vorbereitung für die Wiederverwendung wird die Funktion `reset` des GP aufrufen. Der Destruktor des GP ruft auf allen GP-Erweiterungen, auf die ein Zeiger im GP vorhanden ist, die Funktion `free`[29] auf. Die Funktion `reset` ruft `free` nur auf GP-Erweiterungen auf sog. Auto-Erweiterungen auf (siehe unten) und entfernt dann diese GP-Erweiterungen aus dem GP.

Für GP-Erweiterungen bedeutet dies nun, dass es drei verschiedene Möglichkeiten gibt, diese speichertechnisch zu verwalten:

(vom GP) unabhängige Erweiterung Die Erweiterung wird von einem Modul erzeugt und über `set_extension` dem GP hinzugefügt. Zu einem späteren Zeitpunkt wird die Erweiterung mittels `clear_extension` aus dem GP entfernt. Es obliegt nun dem Modul, die Erweiterung zu löschen oder anderweitig zu verwenden.

Auto-Erweiterung Die Erweiterung wird von einem Modul erzeugt und dem GP über `set_auto_extension` hinzugefügt. Dies bedeutet, dass diese GP-Erweiterung aus dem GP entfernt wird, wenn der Referenzzähler im GP Null erreicht. Zum gleichen Zeitpunkt wird auf dieser Erweiterung die Funktion `free` aufgerufen. Die Funktion `free` muss nun die Erweiterung löschen oder anders geartet zur Wiederverwendung frei geben.

Sticky-Erweiterung Die Erweiterung wird von einem Modul erzeugt und dem GP über `set_extension` hinzugefügt. Zu keinem Zeitpunkt wird `clear_extension` aufgerufen. Dies bedeutet, dass die GP-Erweiterung im GP verbleibt, selbst wenn der Referenzzähler Null erreicht. Sie erhält nur einen Aufruf von `free`, wenn das GP wirklich gelöscht wird. Wird das GP wieder verwendet, ist dann die GP-Erweiterung nach wie vor im GP vorhanden. Einmal hinzugefügt „klebt“ sie also bis zur Zerstörung des GP an diesem.

4.8.2. Anforderungen an GP-Erweiterungen

GP-Erweiterungen entsprechen im Rahmen der taktgenauen busphasenbasierten *J-R*-Simulation verschiedenen Signalen des modellierten MMBIF. Sie können entweder aus dem verhandelbaren oder dem unverhandelbaren Satz von GP-Erweiterungen stammen (siehe Abschnitt 4.6.1).

[29] Diese Funktion `free` ist Teil der Basisklasse `tlm_extension` und nicht mit der Funktion `free` des `tlm_mm_interface` zu verwechseln.

Ein Empfänger eines GP muss es immer bezüglich aller GP-Erweiterungen aus dem unverhandelbaren Satz untersuchen, da dieser diktiert, die GP-Erweiterungen in jedem Fall zu verwenden. Zugriffe auf diese Erweiterungen sollen mit geringer Code-Komplexität und geringen Performancekosten einhergehen.

Ein Sender eines GP muss prinzipiell alle GP-Erweiterungen aus dem unverhandelbaren Satz setzen. Einfache und effiziente Zugriffe auf die Erweiterungen sind auch auf Senderseite notwendig. Jedoch existiert für jedes Signal eines MMBIF ein Defaultwert (siehe Abschnitt 3.2.1), sodass auch einer GP-Erweiterung ein Defaultwert zugeordnet werden kann. Existierte ein Mechanismus, der es erlaubte, festzulegen und festzustellen, ob eine GP-Erweiterung überhaupt verwendet wird, könnte der Sender darauf verzichten, GP-Erweiterungen zu setzen, deren Wert dem Default entsprechen soll. Bei geschickter Festlegung der Defaultwerte bei der Definition der Busphasen kann somit die Code-Komplexität stark reduziert werden, da der Sender nur wenige Nicht-Defaultwerte setzen muss. Für den Empfänger ändert dies nur wenig: Er muss ohnehin, wie oben erwähnt, das GP auf die Erweiterung hin untersuchen. Der Test, ob die Erweiterung überhaupt verwendet wurde, erhöht die Code-Komplexität nicht signifikant.

Für GP-Erweiterungen aus dem verhandelbaren Satz ergeben sich die gleichen Anforderungen, wenn der resultierende Erfordernisgrad zwingend ist. Ist der resultierende Erfordernisgrad verboten, ergeben sich keine Anforderungen, da die GP-Erweiterung nicht verwendet wird. Bei einem resultierenden Erfordernisgrad von optional steht es dem Sender frei, die Erweiterung zu nutzen. Er muss also in der Lage sein, zu markieren, ob die Erweiterung genutzt wird, während der Empfänger dies testen muss. Offenbar wird hier ein ähnlicher Mechanismus benötigt, wie er schon bei den unverhandelbaren Erweiterungen beschrieben wurde. Ist dieser Mechanismus bei den unverhandelbaren Erweiterungen eine Optimierung zur Reduzierung zur Code-Komplexität im Sender, ist er bei den verhandelbaren Erweiterungen aber zwingend notwendig, da im Vorfeld der resultierende Erfordernisgrad nicht klar ist und daher von der Möglichkeit ausgegangen werden muss, das er auf optional gesetzt ist.

Die Anforderungen können also zusammengefasst werden:

1. Mechanismus zur Festlegung und Feststellung, ob eine GP-Erweiterung überhaupt verwendet wird.

2. Einfacher Zugriff auf die Erweiterungen (Code-Komplexität)

3. Gute Zugriffsgeschwindigkeit auf Erweiterungen

4.8.3. Anforderungsanalyse

Laut TLM-2.0-LRM Abschnitt 7.5, Absatz d) ist die Konstruktion von GPs sehr teuer (bezogen auf die Simulationsperformance). Aus diesem Grund wird die Wiederverwendung einmal allozierter GPs stark empfohlen. Bei Verwendung von `nb_transport` wird dazu vorgeschlagen, GP-Pools zu verwenden, aus denen GPs bei Bedarf entnommen und vom MM dort

wieder eingefügt werden. Es ist also vernünftig, bei der Konzeption der Erweiterungen von einem Pool von GPs auszugehen.

Die unabhängigen Erweiterungen sind aus meiner Sicht nicht empfehlenswert. Sie legen eine große Verantwortung in die Hände desjenigen, der die Erweiterung verwendet. Fehlverwendungen wie das zu frühe Entfernen aus dem GP oder gar verfrühtes Löschen der Erweiterung kann zu schwerwiegenden Fehlern bis hin zu Simulationsabstürzen aufgrund von Zugriffen auf ungültigen Speicher führen. Darüber hinaus muss der Anwender viel Erfahrung mit effizienter Speicherverwaltung haben, damit die Simulationsperformance nicht negativ beeinflusst wird.

Unter der Annahme, dass die GPs in einem Pool verwaltet werden (siehe oben), müssen Sticky-Erweiterungen nur einmal erzeugt und einem GP hinzugefügt werden. Danach werden sie vom GP-Pool verwaltet. Sie haben also den geringsten Einfluss auf die Simulationsperformance. Ihre Verwendung ist daher äußerst empfehlenswert. Jedoch gibt eine Sticky-Extension keine Auskunft, ob sie im GP existiert, weil sie gerade hinzugefügt (also verwendet) wurde, oder ob sie im GP existiert, weil sie dem GP im Verlaufe einer vorherigen Transaktion hinzugefügt wurde und nun, nachdem das GP wieder aus dem Pool entnommen wird, noch darin vorkommt. Eine Sticky-Erweiterung allein kann also nicht verwendet werden.

Anders verhält es sich bei Auto-Erweiterungen: Ein GP, das aus dem Pool entnommen wird, enthält nie die Erweiterung, denn wenn der Referenzzähler Null erreicht und das GP zurück in den Pool gelangt, wird die Erweiterung stets entfernt. Dann ist immer klar, dass die Existenz der Erweiterung im GP auch ihre Verwendung anzeigt. Da also Auto-Erweiterungen aus dem GP entfernt werden, wenn der Referenzzähler Null erreicht, müssen sie beim Aufruf von `free()` auch aus Sicht der Sepicherverwaltung wieder freigegeben werden. Wurden sie vor dem Hinzufügen mit Hilfe von `new` erzeugt, müssen sie mit Hilfe von `delete` gelöscht werden, jedoch sind `new` und `delete` sehr teuer aus Sicht der Simulationsperformance, da dabei Speicher aktiv alloziert und wieder de-alloziert wird. Besser ist, die Erweiterungen zur Wiederverwendung frei zu geben. Das heißt, sie sollten vor dem Hinzufügen aus einem Pool entnommen werden und innerhalb von `free()` wieder in den Pool zurückkehren.

Messungen, die diese Aussagen untermauern, wurden z.B. von John Ansley durchgeführt und in [Ansl09]$_i$ (insbesondere Folie 27) präsentiert.

Die obige Diskussion macht deutlich, dass die Auto-Erweiterungen mit Pool eine naheliegende Option für die Implementierung der GP-Erweiterungen im Kontext der taktgenauen busphasenbasierten *J-R*-Simulation mit TLM-2.0 sind. Der Nachteil dabei ist aber, dass die Verwendung des Pools in der Implementierung der Erweiterung zu verankern ist. Das bedeutet, dass es sehr klare Regeln zur Implementierung geben muss. Werden diese nicht eingehalten, wird entweder die Simulationsperformance negativ beeinflusst oder sogar die korrekte Simulation gefährdet. Die Einhaltung solcher Programmierregeln ist schwer automatisch zu überprüfen, die Gefahr für Programmier- und somit Simulationsfehler steigt. Alternativ dazu könnte man eine Basisklasse schaffen, die alle Regeln implementiert und

von der lediglich noch abzuleiten ist. Das Problem dabei ist aber, dass bereits existierende GP-Erweiterungen, die z.B. für die LT-Modellierung implementiert wurden, für die Verwendung im taktgenauen Modell verändert werden müssen. Darüber hinaus ist es der Basisklasse auch nicht möglich, die Verwendung der Speicherverwaltung völlig zu verbergen, da im Rahmen der `clone`-Funktion (siehe Abschnitt E.2) eine neue Instanz der Erweiterung angelegt werden muss.

Mein Fazit aus dieser Analyse ist, dass der Nutzer in der Implementierung der Erweiterung in keiner Weise eingeschränkt werden sollte, weder durch Regeln noch durch Basisklassen. Die Markierung und Abfrage der Gültigkeit einer solchen GP-Erweiterung sollen vollkommen unabhängig von der Implementierung der Erweiterung sein und werden über Zugriffsregeln bestimmt. Die Simulationsperformance von Zugriffen auf die Erweiterung, also die Abfrage oder Markierung der Gültigkeit und das Auslesen oder Setzen des Inhaltes der Erweiterung, soll mit der entsprechenden Performance der Auto-Erweiterung vergleichbar sein. Für die Zugriffsregeln ist eine generische API zu implementieren, die einfach vom Nutzer anzuwenden ist.

4.8.4. Erweiterungskonzept

Wie oben erwähnt soll jede Erweiterung unabhängig von der ursprünglich vorgesehenen Speicherverwaltung verwendbar sein. Die Speicherverwaltung setzt sich einerseits aus der Erzeugung (z.B. Entnahme aus einem Pool oder Erzeugung mit `new`) und andererseits aus der zur Erzeugung inversen Operation in der `free`-Funktion zusammen. Die Implementierung von `free` hängt somit einerseits von der Art der Erweiterung (unabhängige, Auto- oder Sticky-Erweiterung) und andererseits von den Präferenzen und Erfahrungen seitens des Entwicklers bezüglich Speicherverwaltung ab.

Dies bedeutet, dass das Verhalten von `free` als völlig unbekannt angesehen werden muss. Aus diesem Grund muss die Anzahl der Aufrufe von `free` minimiert werden, da `free` im Zweifelsfalle sehr langsam arbeitet (z.B. mit `delete`). Dies kann erreicht werden, wenn die Erweiterung als Sticky-Erweiterung dem GP hinzugefügt wird. Dann wird `free` nur beim Löschen des GP aufgerufen und unter der Annahme, dass GPs in einem Pool verwaltet werden, ist dies nur am Ende der Simulation ein mal der Fall.

Jedoch macht `free` Annahmen über die Art der Erzeugung der Erweiterungen. Die Erweiterung muss also genau so erstellt werden, wie es `free` erwartet, ansonsten wird es beim Aufruf von `free` zu Laufzeitfehlern kommen. Wie aber kann eine Erweiterung auf die korrekte Art und Weise erzeugt werden, wenn man deren Speicherverwaltungskonzept nicht kennt? Listing 4.35 zeigt eine Möglichkeit. Soll eine Erweiterung erzeugt werden, so wird nicht direkt eine Instanz dieser Erweiterung, sondern eine Instanz einer von der Erweiterung abgeleitete Klasse erzeugt. Diese abgeleitete Klasse stellt sicher, dass `free delete` verwendet, da zur Erzeugung `new` verwendet wird. Sollte die Erweiterung mittels `clone` kopiert werden, entsteht eine Instanz der ursprünglichen Erweiterung, die innerhalb von `clone` mit der für die Erweiterung spezifischen Speicherverwaltung erzeugt wird. Wird auf dieser Instanz dann

`free` aufgerufen, handelt es sich um die ursprüngliche `free`-Funktion, und so wird auch die Freigabe korrekt erfolgen.

```
1  template <typename extension>
2  struct override_free_wrapper : public extension
3  {
4    virtual void free(){delete this;}
5  };
6
7  template <typename extension>
8  extension* create_extension(){
9    return new override_free_wrapper<extension>();
10 }
```

Listing 4.35: Erzeugung einer Erweiterung vom Typ `extension` mit Umgehung der erweiterungsspezifischen Speicherverwaltung

Die Erzeugung und Zerstörung mit `new` und `delete` ist für die Sticky-Erweiterung akzeptabel, da pro GP nur eine Sticky-Erweiterung erzeugt wird und dann zusammen mit dem GP im Pool verbleibt. Wird das GP häufig wiederverwendet, werden die Erzeugungskosten für die Sticky-Erweiterung vernachlässigbar. Die einzige Grundannahme ist, dass für die Erweiterung ein Konstruktor ohne Argumente existiert. Diese Voraussetzung kann nicht grundsätzlich als erfüllt angesehen werden, jedoch habe ich in Rahmen meiner Arbeit mit TLM-2.0, insbesondere auch im Rahmen der TLM-WG und meiner Aktivitäten im TLM-2.0-Forum, nie eine Erweiterung gesehen, für die dies nicht zutrifft.

Es ist also möglich, jede beliebige Erweiterung als Sticky-Erweiterung zu nutzen. Jedoch ist es, wie in Abschnitt 4.8.3 beschrieben, damit nicht möglich, die Gültigkeit einer Erweiterung zu setzen oder abzufragen. Dementsprechend muss ein zusätzlicher Mechanismus geschaffen werden. Dieser Mechanismus muss auf Auto-Erweiterungen basieren, da in einem GP bei Entnahme aus dem Pool keine Erweiterung als gültig markiert sein sollte. Dies bedeutet, dass vor dem eigentlichen Zugriff auf die Erweiterung ein zusätzlicher Zugriff auf eine andere Erweiterung notwendig ist, um die Gültigkeit abzufragen. Dies wird die Zugriffszeit negativ beeinflussen. Folglich muss die Speicherverwaltung dieser Erweiterung sehr effizient sein, um die entstehenden Zusatzkosten nicht noch weiter zu erhöhen. Ich nenne die Erweiterung, die die Gültigkeit einer anderen Erweiterung markiert, die **(zugehörige) Schalterweiterung**, da sie die Gültigkeit der anderen Erweiterung an- und abschalten kann.

Zur Markierung der Gültigkeit einer Erweiterung wird die Schalterweiterung dem GP hinzugefügt. Zum Test auf Gültigkeit müssen Empfänger des GP überprüfen, ob die Schalterweiterung im GP vorhanden ist. Es wird deutlich, dass der Inhalt der Schalterweiterung und somit die eigentliche Instanz nicht von Bedeutung sind. Es genügt, eine einzige Instanz einer Schalterweiterungsklasse im gesamten System zu haben. Diese Instanz kann in beliebigen GPs zur Markierung der Gültigkeit der zugehörigen datentragenden Erweiterung verwendet werden. Dadurch ist es möglich, eine Schalterweiterung als Singleton-Objekt auszulegen. Es muss nur einmal erzeugt und niemals zerstört werden. So kann die `free`-Funktion der Schalterweiterung leer bleiben, es sind keinerlei `new`- und `delete`-Aufrufe und auch keine Zugriffe

auf einen Pool während der Simulation nötig. Die Speicherverwaltung der Schalterweiterung ist somit nahezu kostenfrei.

Im Falle von GP-Erweiterungen, deren Änderungsintervall x2x oder p2p ist, darf sich die Gültigkeit ändern[30]. Im oben beschriebenen Konzept bedeutet eine Änderung von gültig auf ungültig, dass die entsprechende Schalterweiterung aus dem GP zu entfernen ist. Es handelt sich dabei aber um eine Auto-Erweiterung, und das TLM-2.0-LRM, Abschnitt 7.20.4 Absatz z) legt fest, dass eine solche GP-Erweiterung nicht mehr manuell entfernt werden darf. Da also die Schalterweiterung nicht entfernt werden kann, muss eine Änderung von gültig auf ungültig auf andere Art und Weise erfolgen.

Es ist möglich, eine Erweiterung im GP zu ersetzen. Auf der im GP vorhandenen Instanz wird `free` aufgerufen, damit diese Instanz freigegeben wird, und an ihre Stelle wird eine neue Instanz gesetzt, die auf die gleiche Art wie die andere Instanz erzeugt wurde[31]. Erkennt man die Existenz der Singleton-Instanz einer Schalterweiterung im GP (d.h. es wird Gültigkeit angezeigt), so kann man diese mit einer anderern Singleton-Instanz ersetzen[32]. Ein Aufruf von `free` ist nicht notwendig, da man weiß, dass `free` dieser Instanz keinen Effekt hat. Zur Änderung auf gültig setzt man dann wieder die erste Instanz in das GP. Dies bedeutet also, dass von der Schalterweiterung genau zwei globale Instanzen existieren: Eine zur Signalisierung der Gültigkeit und eine zur Signalisierung der Ungültigkeit, wenn vorher bereits einmal Gültigkeit gesetzt war.

Der Test auf Gültigkeit ist nun also ein Vergleich des im GP vorhandenen Schalterweiterungszeigers mit dem Zeiger auf die globale Instanz der Schalterweiterung, die Gültigkeit markiert. Ist der Zeiger im GP `NULL`, d.h. die Erweiterung wurde nie als gültig markiert, ergibt der Test, dass die Erweiterung ungültig ist. Stimmt der Zeiger im GP mit dem Zeiger auf die globale Instanz der Schalterweiterung überein, die Ungültigkeit markiert, ergibt der Test auch in diesem Fall, dass die Erweiterung ungültig ist.

Abbildung 4.36 illustriert das Konzept. Im Beispiel gibt es im GP nur eine Erweiterungsklasse mit zugehöriger Schalterweiterungsklasse. Man sieht drei Punkt-zu-Punkt-Verbindungen. Auf allen wird gerade ein GP übertragen[33]. Auf der ersten Punkt-zu-Punkt-Verbindung ist die Sticky-Erweiterung ungültig, da gar kein Zeiger auf die zugehörige Schalterweiterung im GP existiert[34]. Auf der zweiten Verbindung ist die Sticky-Erweiterung gültig,

[30] Eine ungültige Erweiterung stellt im Falle einer unverhandelbaren Erweiterung schließlich den Defaultwert dar.

[31] Man mag jetzt vermuten, man könne nun auch einfach gar keine Instanz im GP platzieren, jedoch wird dies, wenn der Refernzzähler Null erreicht, zu einem Aufruf von `free` auf einer nicht-vorhandenen Erweiterung führen.

[32] Genau genommen handelt es sich nun nicht mehr um Singletons, da genau zwei Instanzen der Erweiterungen im System existieren: Ein Dualton?

[33] Im Beispiel ist es ohne Bedeutung, ob dies das gleiche GP ist oder ob es sich um unterschiedliche GP-Instanzen handelt.

[34] Handelt es sich um eine Erweiterung aus dem unverhandelbaren Satz, so wird damit der Defaultwert der Erweiterung signalisiert. Handel es sich um eine Erweiterung aus dem verhandelbaren Satz, so bedeutet dies, dass das mit der Erweiterung modellierte Signal auf dieser Verbindung effektiv nicht existiert.

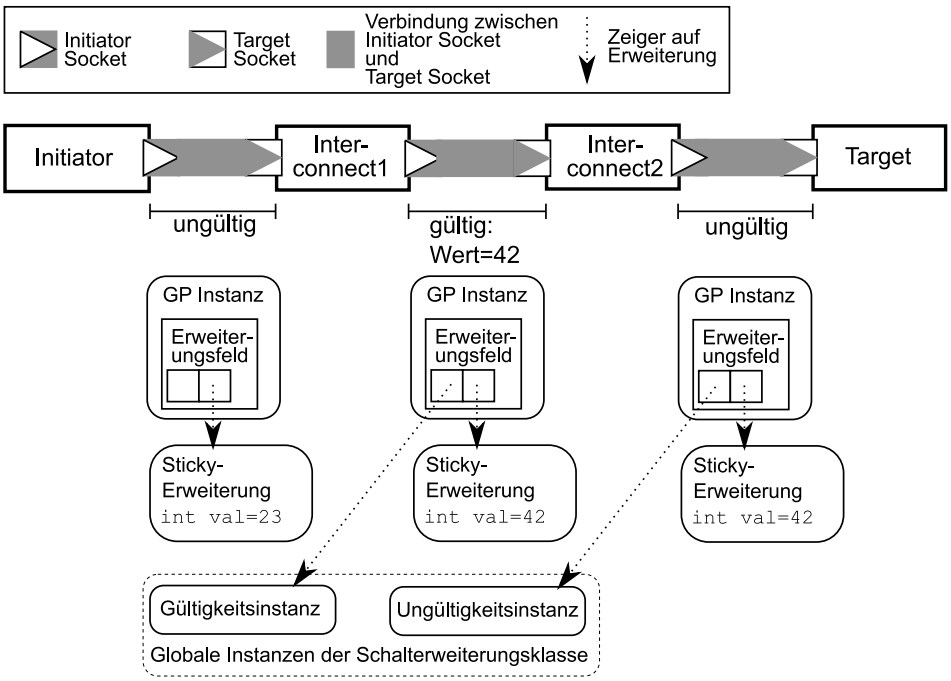


Abbildung 4.36.: Verwendung der globalen Instanzen der zugehörigen Schalterweiterungsklasse zur Markierung der Gültigkeit einer Sticky-Erweiterung

da der Zeiger auf die Schalterweiterung im Erweiterungsfeld des GP auf die Instanz zeigt, die Gültigkeit markiert. Somit wird der Wert aus der Sticky-Erweiterung verwendet. Auf der dritten Verbindung ist die Sticky-Erweiterung wiederum ungültig, da zwar ein Zeiger auf die Schalterweiterung vorhanden ist, dieser aber auf die Instanz zeigt, die Ungültigkeit markiert.

Abbildung 4.37 zeigt das Klassendiagramm einer generischen Schalterweiterungsklasse. Es handelt sich dabei um eine Template-Klasse, mit deren Template-Parameter `ext` die Schalterweiterung für eine beliebige gegebene Erweiterungsklasse ausgeprägt werden kann. Die globalen Instanzen zur Anzeige der Gültigkeit sind als Klassenattribute `valid` und `invalid` ausgeführt. Wie oben beschrieben, ist die Funktion `free` leer. Das gleiche gilt für `copy_from`, da bei der Synchronisierung der Inhalte zweier Instanzen der Schalterweiterung aufgrund der Abwesenheit von Instanzattributen keine Aktivität notwendig ist. Die Funktion zum Duplizieren einer Instanz der Schalterweiterung (`clone`) liefert schlicht einen Zeiger auf die zu duplizierende Instanz zurück. D.h. das Duplikat von `valid` bzw. `invalid` ist wiederum `valid` bzw. `invalid`.

TLM-2.0 wird nun um Listing 4.35 erweitert, damit jede beliebige Erweiterung als Sticky-Erweiterung verwendet werden kann. Auch wird die generische Schalterweiterung zu TLM-2.0 hinzugefügt. Damit ist es möglich, die Zugriffsregeln auf GP-Erweiterungen wie in Abbildung 4.38 auf Seite 100 festzulegen. Mittels `validate` bzw. `invalidate` wird eine Erweiterung als gültig bzw. ungültig markiert. Ablauf `get` liefert den Zeiger auf die Sticky-Erweiterung, während `is_valid` die Gültigkeit überprüft. Abläufe `validate_and_get` und `test_and_get` gruppieren verschiedene Abläufe, da diese Kombinationen sehr häufig verwendet werden.

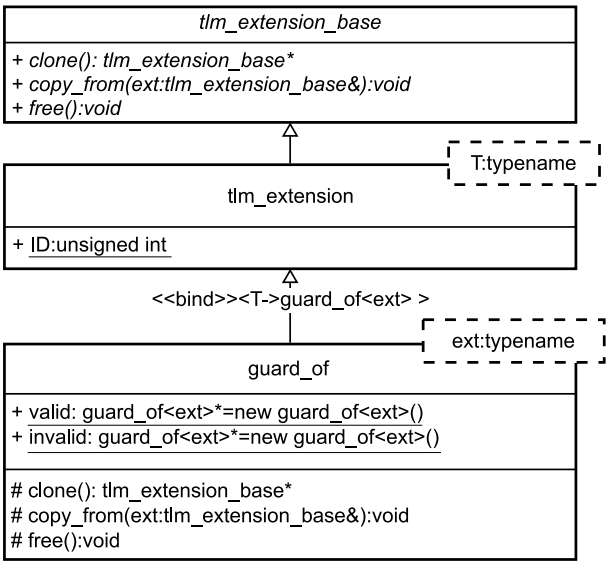


Abbildung 4.37.: Klassendiagramm der generischen Schalterweiterung

Es liegt nahe, diese Abläufe als Funktionen zu implementieren. Dies habe ich getan und diese Funktionen im Namensraum `extension_api` zu TLM-2.0 hinzugefügt. Damit kann dann ein Initiator mittels `extension_api::validate_and_get<my_ext>(gp)->value=42;` die Erweiterung vom Typ `my_ext` im GP `gp` als gültig markieren und gleichzeitig das Attribut `value` der dann im GP vorhandenen Instanz von `my_ext` auf 42 setzen.

Da die Gültigkeitsmarkierung von der eigentlichen Erweiterung abgespaltet und als eigene Auto-Erweiterung implementiert ist, stellt sich die Frage nach der Geschwindigkeit der Zugriffe im Vergleich zu Zugriffen auf eine einzelne, als Auto-Erweiterung entworfene Erweiterung. Um dies zu untersuchen, wurde ein einfaches Experiment durchgeführt (siehe Anhang G). Dabei wurde deutlich, dass die Speicherverwaltung effizienter arbeitet als bei einer einzelnen Auto-Erweiterung und dass das Markieren der Gültigkeit genauso effizient arbeitet wie bei einer einzelnen Auto-Erweiterung. Jedoch benötigt der Zugriff auf eine Erweiterung acht Instruktionen mehr als bei einer einzelnen Auto-Erweiterung. Das Experiment hat aber auch illustriert, dass diese acht Instruktionen unter realistischen Bedingungen die Gesamtzahl von Instruktionen, die bei einer Kommunikation abgearbeitet werden, nur marginal verändert (die Gesamtzahl der Instruktionen liegt in der Regel im hohen dreistelligen Bereich). Die Performance der in Abbildung 4.38 vorgeschlagenen API kann folglich als gut im Vergleich zu einer einzelnen Auto-Erweiterung bewertet werden.

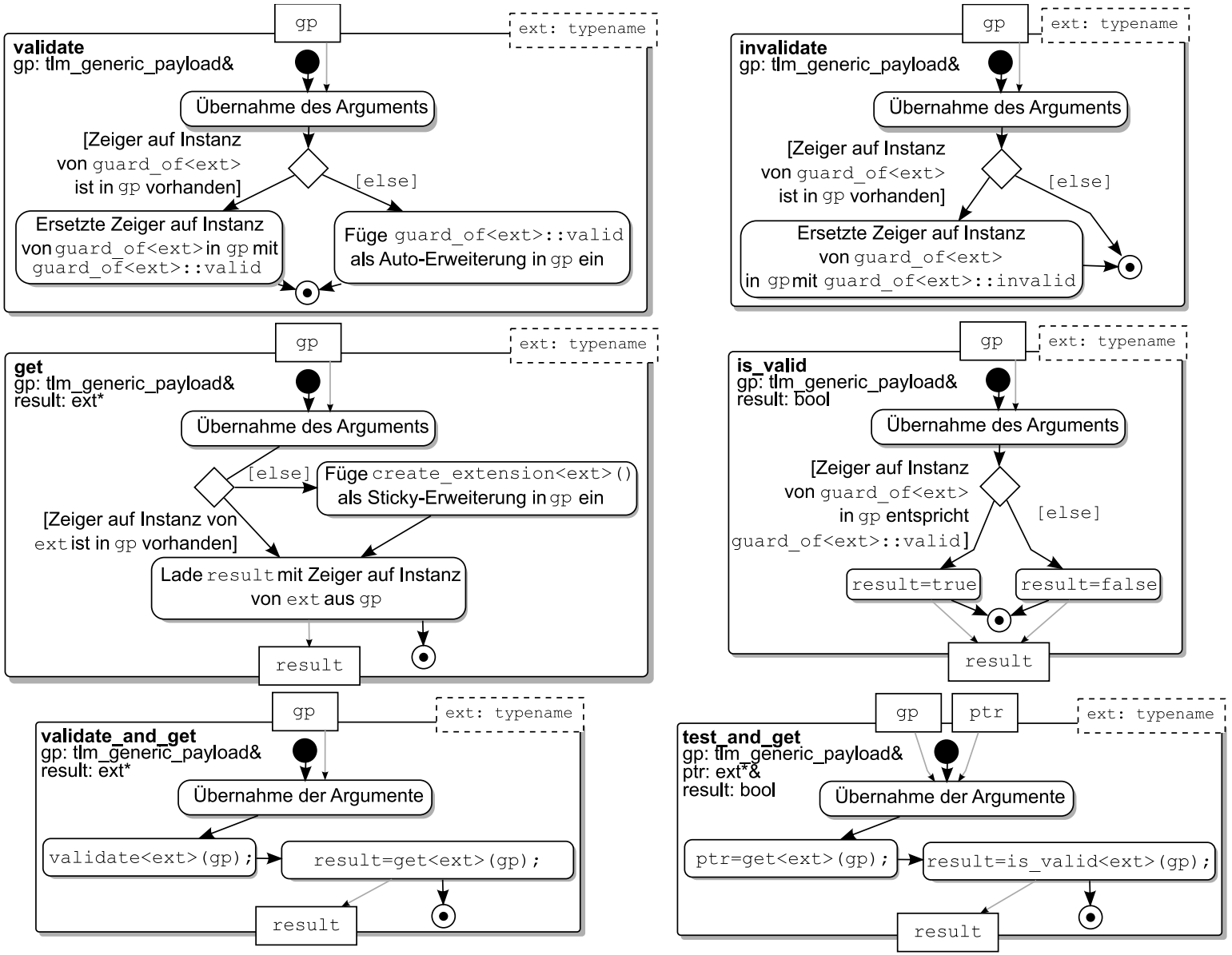


Abbildung 4.38.: Generische API für Erweiterungszugriffe

Zusammenfassend sind die Vorteile der vorgeschlagenen API:

1. Jede beliebige Erweiterung, die einen Konstruktor ohne Argument besitzt, kann verwendet werden. Der Entwickler der Erweiterung kann die Speicherverwaltung seiner Wahl, selbst simples `new/delete` verwenden. Diese Speicherverwaltung kommt nur beim Kopieren von GPs zum Tragen[35].

2. Die Zugriffe sind simpel und benötigen in der Regel nur eine Zeile Nutzer-Code.

3. Der Nutzer muss nie eine Erweiterung manuell instanzieren. Die gesamte Speicherverwaltung ist in der API gekapselt.

4. Die API stellt sicher, dass kein `NULL`-Zeiger in den Nutzer-Code gelangt[36] und vermeidet so Simulationsabstürze in Zusammenhang mit Zugriffen auf nicht-existente Erweiterungen.

5. Die Performance der API ist mit der einer einzelnen Auto-Erweiterung vergleichbar.

Die API vereinfacht also ohne signifikante Performanceeinbußen die Entwicklung und den Umgang mit Erweiterungen.

4.9. Spezielle Konventionen

Das in Abschnitt 4.3 beschriebene GP- und TLM-Phase-Mapping erläutert, wie eine Satz von Busphasen auf die Datenstrukturen von TLM-2.0 abgebildet werden kann. Offen ist dabei noch, welche Phasen mittels `nb_transport_fw` (also von einem Initiatorsocket zu einem Targetsocket) und welche mittels `nb_transport_bw` (also von einem Targetsocket zu einem Initiatorsocket) übertragen werden.

Master werden als Initiatoren modelliert, da sie die Kommunikation starten, und daher wird ein Master-MMBIF als Initiatorsocket modelliert. Entsprechend werden Slave-MMBIF als Targetsockets modelliert. Ein Kommunikationsautomat besitzt demzufolge Targetsockets zur Verbindung mit Mastern und Initiatorsockets zur Verbindung mit Slaves. Busphasen sind eindeutig Mastern oder Slaves zugeordnet (siehe Definition 3.7), und Busphasen haben einen eindeutigen Sender und Empfänger (siehe Abschnitt B.2). Somit gelten folgende Regeln:

1. Die Phase Φ ist einem Master zugeordnet und

1.1 der Master ist Sender: `BEGIN_`Φ und `ABORT_`Φ[37], werden mittels `nb_transport_fw` und `END_`Φ[37] wird mittels `nb_transport_bw` übertragen.

1.2 der Master ist Empfänger: `BEGIN_`Φ und `ABORT_`Φ[37], werden mittels `nb_transport_bw` und `END_`Φ[37] wird mittels `nb_transport_fw` übertragen.

[35] das Kopieren von GPs sollte laut TLM-LRM grundsätzlich auf ein Minimum beschränkt werden, da es eine sehr teure Operation ist.

[36] Im Gegensatz zur Verwendung einer einzelnen Auto-Erweiterung

[37] wenn vorhanden

2. Die Phase Φ ist einem Slave zugeordnet und

2.1 der Slave ist Sender: `BEGIN_`Φ und `ABORT_`Φ^{37}, werden mittels `nb_transport_bw` und `END_`Φ^{37} wird mittels `nb_transport_fw` übertragen.

2.2 der Slave ist Empfänger: `BEGIN_`Φ und `ABORT_`Φ^{37}, werden mittels `nb_transport_fw` und `END_`Φ^{37} wird mittels `nb_transport_bw` übertragen.

Dabei kann anstelle des expliziten `nb_transport`-Aufrufes für `END_`Φ auch der Return-Pfad mittels `TLM_UPDATED` verwendet werden. Die Verwendung des Return-Pfades zur Signalisierung anderer Phasen ist im Rahmen der taktgenauen busphasenbasierten Simulation in keinem Fall ratsam. Meine Untersuchungen mit dem OCP, AMBA und auch CoreConnect haben gezeigt, dass dann die Codekomplexität in den Modellen der Master, Slaves und vor allem Kommunikationsautomaten stark zunimmt, da bei jedem Aufruf von `nb_transport` potentiell jede mögliche noch ausstehende Phasen eintreffen könnte. Zur Vermeidung dieser zunehmenden Code-Komplexität und der daraus resultierenden Fehleranfälligkeit wird daher die Verwendung von `TLM_UPDATED` auf die oben beschriebene Signalisierung von `END_`Φ eingeschränkt.

Endet eine Phase stets im selben Takt, in dem sie startet und hat der Empfänger keine Möglichkeit das Ende zu verzögern (wie z.B. bei Einzeltaktphasen), so muss er immer `TLM_UPDATED` zur Rückgabe von `END_`Φ verwenden. Der Grund dafür ist, dass damit Protokollfehler leichter aufzufinden sind. Wird das entsprechende `BEGIN_`Φ nur mit `TLM_ACCEPTED` beantwortet, so ist klar, dass ein Bearbeitungsfehler seitens des Empfängers vorliegt.

Darüber hinaus darf pro Takt pro Punkt-zu-Punkt-Verbindung pro TLM-Phase nur ein Aufruf von `nb_transport` stattfinden. Das bedeutet, dass ein Aufruf von `nb_transport` mit einer speziellen Phase in einem speziellen Takt nicht widerrufbar oder änderbar ist. Der Aufruf ist endgültig. Nur dann kann `TLM_UPDATED` sinnvoll verwendet werden. Könnten Aufrufe rückgängig gemacht oder nochmals mit geändertem Inhalt auftreten, so müsste jeder vorherige Aufruf reversibel sein. Dies bedeutet, man müsste Änderungen am GP oder am eigenen internen Modulzustand rückgängig machen können. Dazu müssten Backups aller geänderten Werte vorgehalten werden: ein enormer Aufwand, der die Simulationsperformance sehr negativ beeinflusst. Mechanismen, die sicherstellen, dass ein Aufruf endgültig ist, werden in Abschnitt 4.11 aufgezeigt.

Unterstützt ein MMBIF die logische Gruppierung mehrerer Einzelworttransfers (z.B. Burst-Transfers oder Cache-Line-Transfers), so sollte eine solche logische Einheit von Transfers als eine einzige Transaktion und somit mit einem einzigen GP modelliert werden. Dabei muss das Datenfeld des GP groß genug sein, alle Einzelworte zu enthalten, und das GP wird dann mehrfach zwischen Master, Kommunikationsautomat und Slave ausgetauscht, bis alle Einzeltransfers abgeschlossen sind. Wie genau welche Transfers zu gruppieren sind, sollte beim erstmaligen Modellieren des MMBIF dokumentiert werden.

4.10. Taktung

Die taktgenaue busphasenbasierte *J-R*-Simulation mit TLM-2.0 kann nur korrekt arbeiten, wenn alle beteiligten Module ein klares und einheitliches Verständnis des Begriffes des Taktes haben. Es muss sowohl eine standardisierte Definition des Taktbegriffes als auch ein standardisiertes Interface zum Zugriff auf den Takt existieren. Im Folgenden wird der Begriff eines Taktes diskutiert und anschliessend ein Standard dafür gesucht.

4.10.1. Der Taktbegriff

In RTL ist ein Takt eine regelmäßige Event-Folge. Der (simulierte) zeitliche Abstand zwischen den einzelnen Events ist in einer Simulation mit nur einem Takt ohne Bedeutung, denn die Zeitmarke eines Taktes lässt sich bei Kenntnis, um das wievielte Event (**Taktnummer**) es sich handelt, mittels einer Multiplikation mit der Taktfrequenz errechnen. Bei mehreren Takten im System ist lediglich wichtig, dass die Takt-Events der einzelnen Takte dieselbe Vorher-Nacher-Relation erfüllen wie die realen Takte. Die Perioden sind auch dann nicht wichtig.

Basierend darauf können RTL-Simulatoren als sog. Cycle-Based-Simulatoren implementiert werden (z.B. [BuGr07]). Dabei wird nicht mit simulierter Zeit gearbeitet, sondern nur in Taktschritten eines einzigen Basistaktes, von dem alle anderen (langsameren) Takte abgeleitet sind. In einer solchen Cycle-Based-Simulation können dann, wenn die Events, auf die die Prozesse hin gestartet werden, statisch bestimmt sind, die einzelnen Simulationsprozesse auf ein statisches Schedule abgebildet werden, das in jedem Takt einmal abgearbeitet wird. Dies kann deutlich effizienter geschehen als in einer diskreten Event-Simulation mit dynamischen Scheduling (siehe [BuGr07]).

Für taktgenaues TLM ist solch ein Ansatz aber nicht geeignet, da wie in Abschnitt 3.2 beschrieben oft nicht das ganze System als taktgenaues Model implementiert ist. Im Falle einer solchen Mixed-Mode-Simulation müssten dann der Cycle-Based-Simulator und der diskrete Event-Simulator kombiniert werden. Dies erhöht den Aufwand im Simulator und kann dann die Gesamtpeformance reduzieren. Darüber hinaus ist in taktgenauem TLM die Liste der Ereignisse, auf die ein Prozess reagiert, nicht immer statisch bekannt. Taktgenaue TLM-Modelle sind nicht für die Logiksynthese vorgesehen, und dementsprechend sollte der Designer im Gegensatz zu RTL nicht in der Art, wie er das Verhalten modelliert, eingeschränkt werden. Das dynamische Ändern von Ereignissem, auf die ein Prozess reagiert, ist in SystemC üblich und sollte bei taktgenauem TLM nicht verboten werden.

Dementsprechend stellt sich die Frage, wie ein Takt in TLM modelliert werden sollte. Abstraktere Modellierungen, wie der Loosely Timed-Modellierunsstil von TLM-2.0 oder die Bus-Accurate-Abstraktionsebene von GreenBus, verwenden Zeitannotationen, also echte SystemC-Zeitmarken. Ein Takt muss also auch zur Simulationslaufzeit mit Zeitmarken arbeiten, damit abstrakte und taktgenaue Modelle koexistieren können.

Ein Takt in der taktgenauen TLM ist folglich eine Folge von Zeitmarken. Prinzipiell kann dann mit Hilfe einer Division aus einer solchen Zeitmarke errechnet werden, der wievielte Takt es ist. In einer Simulation, sei es aufgrund der Existenz mehrerer Takte oder aufgrund von Mixed-Mode-Simulationen, kann es dazu kommen, dass Kommunikationen mit einer Zeitmarke auftreten, die kein ganzzahliges Vielfaches des Taktes sind, der der Kommunikation zugeordnet ist. Es muss also eine eindeutige Zuordnung von Zeitmarken zu Taktnummern existieren:

Definition 4.39 :
Es seien eine Taktperiode $p = n * 10^b$ und zwei Zeitstempel $t = m * 10^b$ und $s = o * 10^b$, mit $n, m, o \in \mathbb{N}, 0 < n < 2^{64}, 0 \leq m < 2^{64}, 0 \leq o \leq m$ und $b \in \{-12, -9, -6, -3, 0\}$ in Sekunden gegeben. Dann ermittelt sich die **Zeitmarken-Taktnummern-Zuordnung**, also die Taktnummer $c \in \mathbb{N}$ des zur Zeitmarke s gestarteten, frequenzstabilen, nicht stoppbaren Taktes mit der Periode p, der t zugeordnet wird, mit $c = \lfloor \frac{t-s}{p} \rfloor + 1$. Oder in Intervalschreibweise: Alle Zeitmarken im Interval $[(c-1) * p + s, c * p + s)$ werden der Taktnummer $c \in \mathbb{N}, c > 0$ zugegeordnet.

In Definition 4.39 wird explizit von einem nicht stoppbaren, frequenzstabilen Takt gesprochen. Das bedeutet, dass der Takt nicht abgeschaltet werden kann und immer die gleiche Periode hat, also ununterbrochen in gleichen Abständen „tickt“. Ein Stopp eines Taktes kann als Frequenzänderung auf Null Hertz und dementsprechend ein Start als eine Frequenzänderung von Null auf eine größere Frequenz angesehen werden. Ein stoppbarer, frequenzvariabler Takt kann also eine Folge von Frequenzänderungen erfahren. Zwischen zwei solcher Änderungen kann der Takt als nicht stoppbar und frequenzstabil angesehen werden. Ist für einen Zeitstempel x die Liste der Zeitstempel aller bisherigen Frequenzänderungen eines Taktes $\{t_1, t_2, ..., t_n\}$ mit $t_1 \leq t_2 \leq ... \leq t_n \leq x$ bekannt, so bestimmt sich die Taktnummer für Zeitstempel x wie folgt.

Unter Verwendung von $t_{n+1} = x$ ergibt sich die Taktnummer für Zeitstempel x, indem man die Taktnummern für alle t_X mit $2 \leq X \leq n + 1$ nach Definition 4.39 bestimmt, wobei stets $s = t_{(X-1)}$ gesetzt wird und p der zu $s = t_{(X-1)}$ angenommenen Taktperiode entspricht[38] und diese aufsummiert. Anschaulich bedeutet das, dass mit jeder Frequenzänderung ein neuer Takt „entsteht“, der zum Änderungszeitpunkt startet, dessen Taktnummer aber nicht bei Null beginnt, sonder bei der Taktnummer, die der Takt bis dahin erreicht hatte.

Im Folgenden wird untersucht, wie eine solche Zeitmarkenfolge modelliert werden kann.

4.10.2. Überblick über existierende Taktmodellierungen

Bei meinen Untersuchungen bezüglich der Modellierung eines Taktes für taktgenaues TLM habe ich verschiedene existierende Ansätze gefunden. Diese werden im Folgenden gelistet und anschließend analysiert.

[38] Mit $p = 0$ führt dies zu einer Division durch Null, daher wird in diesem Fall die Taktnummer für t_X auf Null definiert. Dies drückt aus, dass ein abgeschalteter Takt nicht tickt.

Taktmodellierung mit `sc_clock`

Die naheliegendste Modellierung ist die Verwendung des `sc_clock` aus der SystemC-Bibliothek. Diese erzeugt pro Takt zwei Events, die positive Flanke und die negative Flanke, und hat einen binären Wert, der mit jeder Flanke entsprechend wechselt. Die CoWare-SystemC-Modeling-Library [CoWa09]$_i$ erweitert die `sc_clock` um die Möglichkeiten den Takt zu (de-)aktivieren und während der Simulation die Frequenz zu ändern. Darüber hinaus stellt sie noch Taktfrequenzteiler zur Verfügung.

Taktmodellierung mit zeitbehaftetem Warten

Eine weitere Art der Taktmodellierung ist die Verwendung der Taktperiode für modulinterne zeitbehaftete `wait`- oder `next_trigger`-Anweisungen. Dabei gibt es keine zentrale Entität, die Taktereignisse zu bestimmten Zeitmarken erzeugt, sondern lediglich eine zentralisierte Informationsquelle für die Taktperiode eines Taktes ([ScBr06]$_i$). Jedes Modul bekommt von dieser Entität die aktuelle Taktperiode mitgeteilt und kann dann selbst die modulinternen Prozesse zu den entsprechenden Zeitpunkten aktivieren. Mit Hilfe einfacher Multiplikationen mit dieser Taktperiode können dann auch inaktive Takte übersprungen werden. Die zentrale Entität kann die Module auch über Änderungen der Taktperiode informieren.

Taktmodellierung nach Grellier

In [Grel08] analysiert Grellier die Anforderungen an ein Taktmodell, definiert einen Vorschlag für ein Standard-Taktmodell und präsentiert eine eigene, äußerst effiziente Implementierung dieses Standards. Grellier identifiziert für die Modellierung komplexer Taktbäume vier Primitive: Taktquellen, -frequenzteiler, -frequenzvervielfacher, und -multiplexer.

```
1  struct ti_clock_wait_if : virtual sc_interface {
2    virtual sc_event& posedge_in_cycle(unsigned latency) = 0;
3    virtual sc_event& negedge_in_cycle(unsigned latency) = 0;
4
5    virtual void stop()    = 0;
6    virtual void restart() = 0;
7
8    virtual unsigned long long num_ticks() const = 0;
9    virtual  const sc_time& period() const = 0;
10   virtual  double duty_cycle() const = 0;
11   virtual  sc_time start_time() const = 0;
12
13   virtual sc_event& get_period_changed_event() = 0;
14 };
```

Listing 4.40: Takt-Basisinterface von Grellier

Mit diesen kann die Taktverteilung in komplexen SoCs modelliert werden. Alle vier Primitive haben individuelle Interfaces, die sich aber in großen Teilen überschneiden. Es gibt also ein gemeinsames Basisinterface. Grellier definiert dies wie in Listing 4.40 dargestellt. Die IMCs `posedge/negedge_in_cycle` erlauben es, ein Taktereignis, das `latency` Takte in der

Zukunft liegt, anzufordern. Dies erlaubt es, dass feste Verzögerungen innerhalb eines Moduls nicht durch Abzählen von Takten erfolgen, sondern indem der Takt das entsprechend abgezählte Ereignis liefert. Dies zentralisiert die Abzählung von Taktereignissen und erlaubt es einem Takt darüber hinaus auch Taktereignisse zu identifizieren, an denen kein einziges Modul interessiert ist.

Die IMCs `stop` und `restart` erlauben es, ein Taktbaum-Primitiv zu starten und zu stoppen. Die IMCs `num_ticks`, `period`, `duty_cycle` und `start_time` liefern die aktuelle Taktnummer, die Taktperiode, das Tastverhältnis bzw. den Startzeitpunkt des Taktes. `get_period_changed_ event()` liefert das Ereignis, welches bei einer Änderung der Taktperiode auftritt.

Zusätzlich dazu haben die vier Taktbaum-Primitive jeweils die in Listing 4.41 dargestellten IMCs. Für eine Taktquelle kann die Frequenz geändert werden, bei Taktfrequenzteilern bzw. -vervielfachern kann der Wert für die Teilung bzw. Vervielfachung gesetzt werden, während bei einem Taktmultiplexer der Takt gewählt werden kann, der an den Ausgang des Multiplexers gelangen soll.

```
1  struct ti_clock_if : virtual ti_clock_wait_if {
2    virtual void set_period(const sc_time& period_, double duty_cycle_ = 0.5) = 0;
3    virtual void set_period(double period_v_, sc_time_unit period_tu_, double duty_cycle_) = 0;
4    virtual void set_frequency(unsigned frequency, double duty_cycle_ = 0.5) = 0; // in MHz
5  };
6  struct ti_freq_divider_if : virtual ti_clock_wait_if {
7    virtual void set_divider(unsigned divider) = 0;
8    virtual unsigned get_divider() const = 0;
9  };
10 struct ti_freq_multiplier_if : virtual ti_clock_wait_if {
11   virtual void set_multiplier(unsigned multiplier) = 0;
12   virtual unsigned get_multiplier() const = 0;
13 };
14 struct ti_clock_mux_if : virtual ti_clock_wait_if {
15   virtual void set_source(unsigned source_index) = 0;
16   virtual ti_clock_wait_if* get_source() const = 0;
17 };
```

Listing 4.41: Interfaces der Taktbaum-Primitive von Grellier

Analyse

Das selektive Abschalten von Teilen eines SoCs ist in modernen Designs ein wichtiger Faktor, um die Verlustleistung zu minimieren ([Pedr01]). Dies geschieht oft, indem man Teile der Taktverteilung deaktiviert. Auch das dynamische Ändern der Taktfrequenz wird in modernen SoC zur Steuerung der Leistungsaufnahme genutzt. Diese Aktivierungen, Deaktivierung und Änderungen haben einen Einfluss auf die Performance des Systems und müssen z.B. im Rahmen der Performance-Evaluation und natürlich auch im Rahmen der Power-Analyse berücksichtigt werden. Da sowohl Performance-Evaluation als auch Power-Analyse wichtige Anwendungen von taktgenauem TLM und somit auch der taktgenauen busphasenbasierten *J-R*-Simulation mit TLM-2.0 sind, müssen diese Eigenschaften eines Taktes modelliert werden. Aus diesem Grund ist die direkte Verwendung von `sc_clock` nicht geeignet. Die von

CoWare erweiterte `sc_clock` überwindet zwar diesen Nachteil, bietet aber keine Möglichkeit der Taktabzählungszentralisierung, wie es Grellier erlaubt. Wie Experimente von Grellier und auch von mir (siehe Kapitel 5.5) zeigen, ist der Ansatz des zeitbehafteten Wartens mit steigender Anzahl von Prozessen nicht empfehlenswert.

Der aus meiner Sicht beste Ansatz ist somit die Taktmodellierung nach Grellier, jedoch kann er für die taktgenaue busphasenbasierte Simulation noch überarbeitet werden. Meine Erfahrungen und Diskussionen in verschiedenen Working-Groups (vorrangig die OCP-SLD-WG) haben mir gezeigt, dass die Existenz der negativen Taktflanke im Modell oft nicht notwendig ist. Diese als integralen Teil des Taktinterfaces mitzuführen (`negedge_in_cycle`), erachte ich als nicht notwendig. Auch die feste Verankerung der vier von Grellier identifizierten Taktbaumprimitive in einem Taktstandard ist meiner Meinung nach nicht notwendig. Weitere Primitive (wie ein Phasenschieber) sind denkbar und es können nie alle in einem Standard erfasst werden. Der Zugriff auf diese Primitive ist nur für einen Clock- oder Power-Controller notwendig. In der Regel kommen solche Controller und die Taktteiler, -multiplizierer oder -multiplexer vom selben Hersteller. Ein Standardinterface ist dann nicht notwendig.

Darüber hinaus ist die Anzahl der in einem Taktbaum vorhandenen Taktteiler, -multiplizierer oder -multiplexer oft sehr gering. Selbst in komplexen Systemen mit hunderten von Modulen sind dies oft weniger als zehn. Sollten diese auch aus unterschiedlichen Quellen stammen, sind die Interfaces zwischen den Controllern und Taktteilern, -multiplizierern und -multiplexern so trivial (siehe Listing 4.41), dass die wenigen notwendigen Adapter extrem einfach und schnell zu implementieren sind.

4.10.3. Takt-Interfaces für taktgenaue busphasenbasierte TLM-Simulation

Wie im vorangegangenen Abschnitt erläutert, soll ein Taktinterface verwendet werden, das an das von Grellier definierte angelehnt ist. Das dafür vorgeschlagene Interface `tlm_clock_if` wird in Abbildung 4.42 dargestellt. Da die negative Flanke eines Taktes nicht modelliert wird, gibt es nur noch ein Event pro Takt, welches mittel `edge_in_cycle` abgerufen werden kann. Aus dem gleichen Grund gibt es auch keinen IMC `duty_cycle`. Die Bedeutung der `regsiter_as_...` Methoden wird weiter unten klar.

Mit Hilfe dieses Interfaces können Modelle für sie nicht relevante Takte sehr einfach überspringen. Ein Takt ist dann stets drüber informiert, ob eine Taktflanke überhaupt relevant ist und kann diese dann ggf. vollständig auslassen.

Jedoch reicht dieses Interface allein für einen Takt nicht aus. Betrachten wir einen Taktmultiplizierer, der einen Quelltakt mit Faktor N multiplizieren soll. Das heißt der Multiplizierer fügt zwischen zwei Flanken des Quelltaktes $N - 1$ äquidistante Flanken ein (die er dann wiederum überspringen kann, falls keiner diese braucht). Jedoch teilt der Quelltakt nicht mit, wann er stoppt. Damit dies vom Multiplizierer erkannt werden kann, muss dieser jede Taktflanke des Quelltaktes abfragen: Wenn der Quelltakt eine Flanke erzeugt hat, kann der

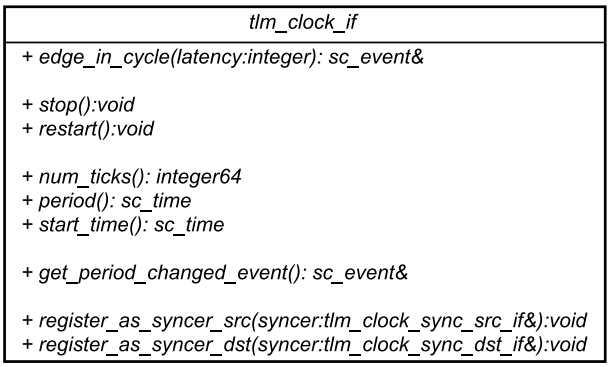


Abbildung 4.42.: TLM-Takt-Interface

Mutliplizierer seine $N-1$ Flanken erzeugen und muss dann wieder die nächste Flanke des Quelltaktes abfragen, um einen evetuellen Stopp des Quelltaktes anhand der Abwesenheit der nächsten Quellflanke zu erkennen. Somit kann der Quelltakt nie eine Flanke überspringen, da der multiplizierte Takt an jeder Quellflanke interessiert ist. Eine Event-basierte Mitteilung eines Stopps ist nicht möglich, da dann eine schwer zu lösende Race-Condition zwischen der Taktflanke des multiplizierten Taktes und dem Stopp-Event auftritt[39]. Aus diesem Grund braucht man zusätzliche Interfaces für eine derartige Synchronisation von Takten, die nicht den Event-Mechnismus nutzen. Mein Konzept zur Synchronisation zweier Takte, also das Ableiten eines Taktes aus einem anderen, wird in Abbildung 4.43 gezeigt.

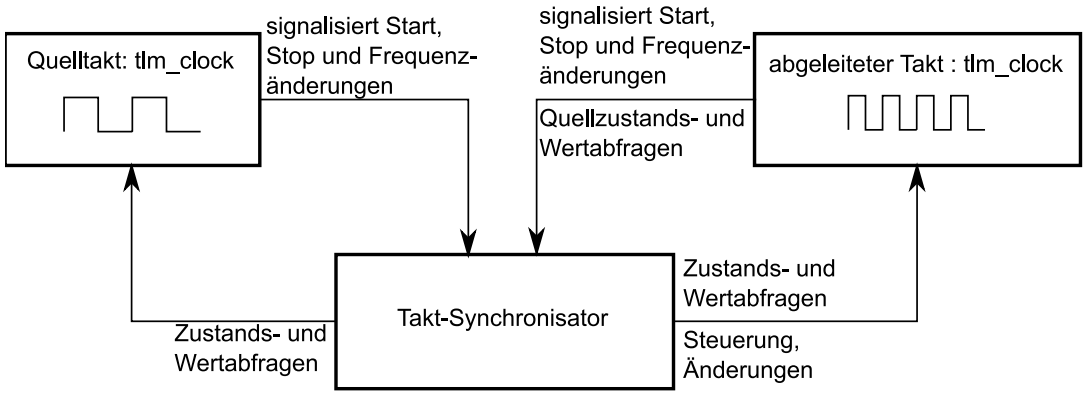


Abbildung 4.43.: Konzept zur Taktsynchronisation

Der Quelltakt und der abgeleitete Takt in Abbildung 4.43 sind zwei gleichartige, eigentlich unabhängige Takte. Diese sind aber über eine Takt-Synchronisierer verbunden. Der Quelltakt teilt jeden Start, Stopp und jede Frequenzänderung an den Synchronisierer mit. Dieser kann dann wiederum den Zustand des Quelltaktes abfragen und dementsprechend auf den

[39]Stoppt der Quelltakt genau zum Zeitpunkt seiner Flanke, wobei er aber die Flanke unterdrückt, will der multiplizierte Takt zu genau diesem Zeitpunkt aber auch eine Flanke erzeugen, welche aber aufgrund des Quell-Stopps unterdrückt werden muss. Es ist dann schwer sicherzustellen, dass das Stopp-Event vor der Erzeugung der Flanke im multiplizierten Takt eintrifft.

abgeleiteten Tatkt einwirken (starten, stoppen, Frequenz ändern). Auch der abgeleitete Takt teilt Änderungen dem Synchroniserer mit, damit dieser über den Zustand des abgeleiteten Taktes auf dem Laufenden bleibt. Darüber hinaus kann der abgeleitete Takt auch den Quellzustand abfragen, z.B. wenn der abgeleitete Takt nach seiner Periode befragt wird, welche vom Quelltakt abhängt.

Der Vorteil bei diesem Vorgehen ist, dass beide Takte nie miteinander kommunizieren müssen. Der Synchronisierer wird nur im Falle von Starts, Stopps und Frequenzänderungen aktiv. Dadurch können beide Takt immer die weitest möglichen „Sprünge“ relativ zu ihrer Taktperiode machen.

Eine Instanz der Taktklasse (in Abbildung 4.43 `tlm_clock` genannt) ist sich nicht darüber bewusst, ob sie ein Quelltakt oder ein abgeleiteter Takt ist. Die Signalisierungen von Start, Stopp und Frequenzänderungen zu einem Takt-Synchronisierer hin erfolgen immer, unabhängig davon, ob dieser wirklich existiert. Dies bedeutet, dass ein Default-Synchronisierer existieren muss, der die entsprechenden Zugriffe verarbeitet, wenn keine wirkliche Synchronisation stattfindet. Der grundsätzliche Aufbau und die Interfaces eines Synchronisierers sind in Abbildung 4.44 dargestellt.

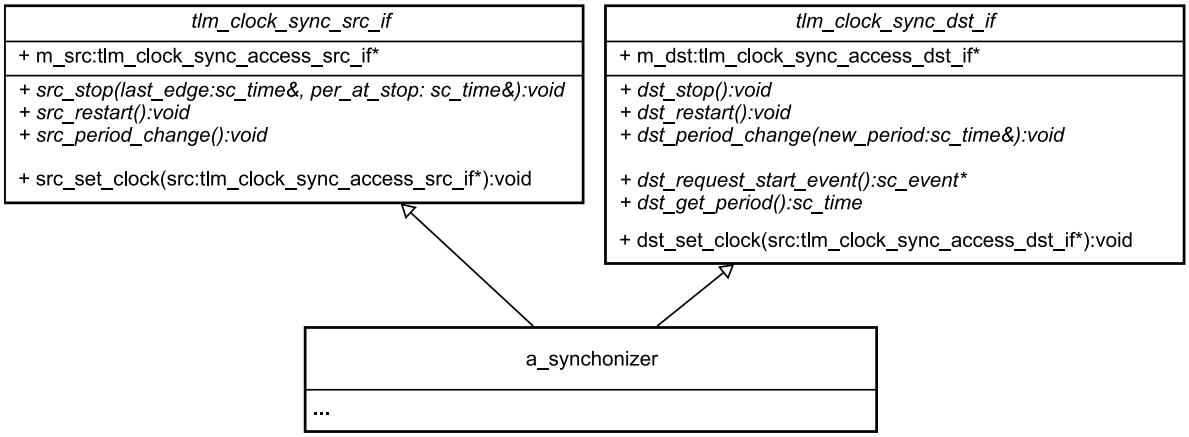


Abbildung 4.44.: Klassendiagramm für einen TLM-Takt-Synchronisierer

Das Interface `tlm_clock_sync_src_if` erlaubt es einem Takt, Informationen als Quelltakt an einen Takt-Synchronisierer zu übergeben. Dabei kann der Takt seinen Stopp, Start oder eine Änderung seiner Periode mitteilen[40].

Das Interface `tlm_clock_sync_dst_if` erlaubt es einem Takt, Informationen als abgeleiteter Takt an einen Takt-Synchronisierer zu übergeben und Informationen abzufragen. Dabei kann der Takte einen Stopp, Start oder eine Änderung seiner Periode mitteilen sowie den Synchronisierer nach seiner Taktperiode und einem Event, dass den Takt startet fragen[40].

Damit der Synchronisierer auf Quell- und abgeleiteten Takt zugreifen kann, müssen auch Takte spezielle Interfaces dafür anbieten. Abbildung 4.45 zeigt das Klassendiagramm für einen TLM-Takt. Neben der Ableitung vom `tlm_clock_if` sind auch die Interfaces

[40]Die einzelnen Funktionen werden im Detail in Anhang H erläutert.

`tlm_clock_sync_access_src_if` und `tlm_clock_sync_access_dst_if` zu erkennen. Diese erlauben einem Synchronisierer den Zugriff auf einen Takt als Quelltakt bzw. als abgeleiteten Takt.

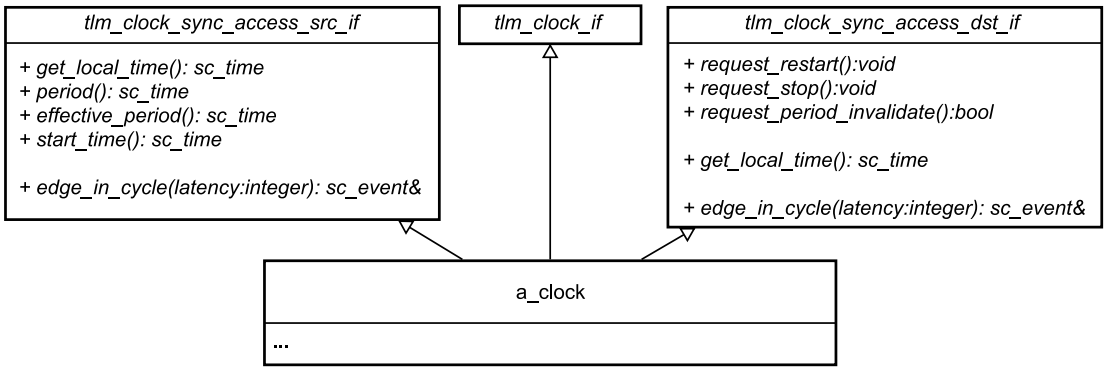


Abbildung 4.45.: Klassendiagramm für einen TLM-Takt

Über das Interface `tlm_clock_sync_access_src_if` kann ein Synchronisierer auf einen Quelltakt zugreifen. Dabei kann er den Zeitstempel der letzten simulierten Flanke, die Periode, die Startzeit und auch Flanken-Events abrufen[40].

Über das Interface `tlm_clock_sync_access_dst_if` kann ein Synchronisierer auf einen abgeleiteten Takt zugreifen. Dabei kann er Starts, Stopps oder Periodenänderungen auslösen sowie den Zeitstempel der letzten simulierten Flanke und Flanken-Events abrufen[40].

Um zwei Takte über einen Synchronisierer zu koppeln, bietet das `tlm_clock_if` (Abbildung 4.42) die Funktionen `register_as_syncer_src` bzw. `register_as_syncer_dst`. Die Implementierung der Funktionen innerhalb eines Taktes muss auf dem jeweils übergebenen Argument `src_set_clock` bzw. `dst_set_clock` aufrufen, damit bi-direktionale Kommunikation zwischen dem Synchronisierer und dem Takt möglich ist.

Mit Hilfe dieser Interfaces habe ich eine effiziente Taktklasse implementiert und dazu auch noch Synchronisierer für Taktteilung, -vervielfachung, -multiplexing und für die Erzeugung einer negativen Flanke (siehe Kapitel 5.5).

4.11. Partielle Prozessausführungsordnung innerhalb eines Taktes

Wie in Abschnitt 4.9 erläutert, darf pro Takt pro Punkt-zu-Punkt-Verbindung pro TLM-Phase nur ein Aufruf von `nb_transport` stattfinden. Dies ist bei Kommunikation, die nur vom synchron ausgelesenen Zustand (inklusive des Zustands seiner Kommunikationsverbindungen) des kommunizierenden Moduls ausgeht, sehr einfach: Ein Modul kann einfach sicherstellen, dass seine getakteten Prozesse vor allen eingehenden Kommunikationen abgearbeitet werden. So arbeiten die getakteten Prozesse nur auf den Werten der Modul-Attribute und auch auf den Zuständen der Kommunikationsverbindungen, die zum Ende des letzen Taktes

vorhanden waren, also genau so, wie es bei synchronen Datenzugriffen gewünscht ist. Da die getakteten Prozesse nur genau einmal ausgeführt werden, sind alle durch sie ausgelösten Kommunikationen endgültig.

Schwieriger ist Kommunikation, die auf Events innerhalb eines Taktes (inklusive Kommunikationsereignissen) hin noch im selben Takt ausgeführt wird. Basiert diese Kommunikation auf mehr als einem unabhängigen Event, gibt es eine Entscheidungsproblematik: Ein Event existiert innerhalb eines Taktes oder nicht. Tritt eines auf, ist nicht klar, ob die anderen Events im gleichen Takt noch auftreten werden oder ob sie überhaupt nicht auftreten. Auf diese Events zu warten (falsch, wenn sie nicht in diesem Takt auftreten), kann genauso falsch sein, wie direkt auf das empfangene Event hin zu kommunizieren (falsch, wenn später im Takt noch ein anderes Event eintrifft, dass die getätigte Kommunikation als falsch bewertet).

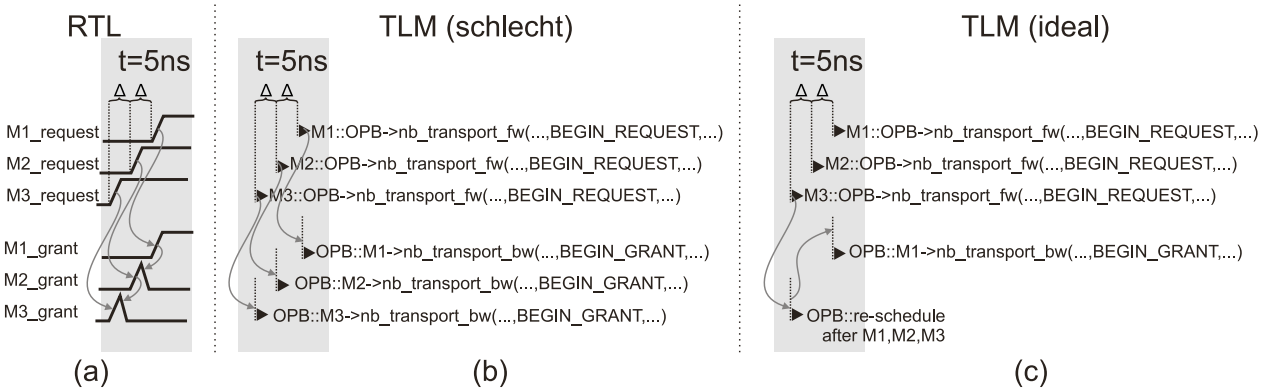


Abbildung 4.46.: Kombinatorische Arbitrierung des OPB in RTL und TLM

Abbildung 4.46 zeigt dazu als Beispiel die Simulation der kombinatorischen Arbitrierung im IBM CoreConnect OPB [IBM-01] in einem DES, sowohl in RTL (Abbildung 4.46a) als auch in TLM (Abbildung 4.46b und c). Alle Master starten ihre Requests zum selben Simulationszeitpunkt, aber in unterschiedlichen Deltazyklen (Δ). In der RTL-Simulation lösen die Signaländerungen auf den Request-Leitungen Events aus, auf die hin die Ergebnisse auf den Grant-Leitungen berechnet werden. Im gezeigten Beispiel hat `M1` die höchste und `M3` die niedrigste Priorität. Da aber `M3` zuerst einen Request absetzt, erhält er auch (für einen Deltazyklus) ein Grant. Mit dem Eintreffen eines höher-prioren Requests stelllt das RTL-Modell diese Fehlentscheidung fest und revidiert sie. Ein solches vorübergehendes falsches Signal nennt man auch Deltazyklus-Glitch, ein nicht ungewöhnlicher Effekt in einer RTL-Simulation.

Eine schlechte TLM-Simulation (Abbildung 4.46b) würde genauso arbeiten, also falsche IMC (im Beispiel `nb_transport_bw` mit `BEGIN_GRANT`) absetzen. Nun muss das TLM-Modell die Aufrufe auch rückgängig machen. Dies bedeutet einerseits ein erhöhtes Kommunikationsaufkommen und andererseits muss der Empfänger dann alle ggf. am GP vorgenommenen Änderungen rückgängig machen. Beides verkompliziert den Code des Modells und belastet den Simulator und bremst somit die Modellentwicklung und die Simulation. Eine ideale TLM-Simulation (Abbildung 4.46c) muss in der Lage sein, bei Auftreten eines Ereignisses

die Ausführung des Prozesses, der eine kombinatorische Berechnung durchführen will, solange zu verzögern, bis alle Prozesse, die Eingaben für die Berechnung liefern können, sicher abgearbeitet wurden. Dabei kann „abgearbeitet“ auch bedeuten, dass der Prozess im jeweiligen Takt überhaupt nicht ausgeführt wird. Man benötigt also eine partielle Ordnung für die Ausführungsreihenfolge von Prozessen. In der diskreten Event-Simulation ist die Ausführungsreihenfolge von Prozessen aber zufällig.

Es muss also ein Mechanismus gefunden werden, der diese Ordnung ermöglicht. Innerhalb eines Moduls kann der Modulentwickler selbst für diese Ordnung sorgen, er hat aber keinen Einfluss auf das Auftreten von Kommunikationsereignissen. Dazu benötigt er Informationen und Regeln, wie diese Informationen genutzt werden können, um die Ordnung herzustellen. Innerhalb der OCP-IP-SLD-WG wird dazu die sog. **Timing-Information-Distribution (TID)** verwendet. Diese hat sich gegenüber anderen Ansätzen als gut geeignet erwiesen, da sie eine gute Simulationsperformance und einen geringen Code-Overhead hat [GüKA07] und dabei mit einem Standard-SystemC-Kernel arbeiten kann. Der einzige Nachteil liegt darin, dass Simulationszeit-Artefakte entstehen. Details dazu werden in Abschnitt 5.4 erläutert.

Das Konzept der TID ist einfach. Angenommen jedes Modul kennt die Zeitdifferenz nach dem Auftreten der Taktflanke, nach der ein Kommunikations-Event spätestens stattgefunden hat. Diese Zeitdifferenz sei als eine Zeitmarke $n \in [0, t_{Taktperiode})$ für jede eingehende Kommunikation bekannt. Dann kann jedes Modul ein Minimum n_{min} und ein Maximum n_{max} für die an einer kombinatorischen Berechnung teilnehmenden, eingehenden Kommunikationen bestimmen. Reagiert der Prozess, der die kombinatorische Berechnung durchführt, mit Hilfe von Events auf eine eingehende Kommunikation, kann er in der Regel nicht bestimmen, welches Event (also welche Kommunikation) ihn gestartet hat. Somit legt er seinen Neustart (bei dem dann die Berechnung durchgeführt wird) um die Zeit $n_{max} - n_{min} + t_{base}$ in die Zukunft, wobei t_{base} die kleinstmögliche Zeitmarke des Simulator ist. Damit ist sicher gestellt, dass die Berechnung erst ausgeführt wird, wenn alle eingehenden Kommunikationen spätestens stattgefunden haben. Das späteste Erzeugen des Ergebnisses der kombinatorischen Berechnung ist dann mit $2 * n_{max} - n_{min} + t_{base}$ gegeben. Diese Information brauchen Module, die mit dem Ergebnis selbst wieder kombinatorische Berechnungen anstellen wollen.

Wie Anhang C zeigt, gibt es Module, deren Eingangskommunikation einen anderen Takt hat als deren getaktete Prozesse und deren Ausgangskommunikation. Nehmen wir an, dass der Eingangstakt doppelt so schnell arbeitet wie der eigene Takt, dann können während einer Ausgangstaktperiode (also einer Periode des eigenen Taktes) zwei Eingangskommunikationen mit der gleichen Phase und gleicher Transaktion empfangen werden, wobei nur die letzte auch eine Ausgangskommunikation zur Folge haben darf, da sonst zwei Ausgangskommunikationen mit gleicher Phase und gleicher Transaktion innerhalb eines Ausgangstaktes auftreten können. Man kann also sagen, dass ein Modul, das einen kombinatorischen Pfad von einer Taktdomäne in die andere hat, die Existenz einer synchronen Datenübernahme am Ende des kombinatorischen Pfades vorwegnimmt und die für diese Synchronisation ohnehin unsichtbaren Änderungen gar nicht erst weiterleitet.

Auch dafür kann die TID eingesetzt werden. Dazu wird für die Eingangskommunikation, wie oben beschrieben, die Zeit bestimmt, nach der die Eingangskommunikation sicher abgeschlossen ist. Dann wird das spätestmögliche Auftreten einer Eingangstaktflanke im Ausgangstakt berechnet. Zur Berechnung eines Ergebnisses verzögert sich der Prozess bis zur letzten Eingangstaktflanke innerhalb eines Ausgangstakts[41] und danach noch bis zu dem Zeitpunkt, wenn alle Eingangskommunikationen abgearbeitet sind. Dann braucht auch eine solche kombinatorische Berechnung zwischen zwei Taktdomänen nur genau einmal während einer Ausgangstaktperiode durchgeführt werden. Ein Modul, das eine solche Berechnung durchführt, muss auch dementsprechend den spätestmöglichen Zeitpunkt seiner Berechnung den folgenden Modulen mitteilen.

Damit dieser Ansatz funktioniert, müssen Module in der Lage sein, die entsprechenden Informationen auszutauschen, diese Information zu verstehen und zu respektieren. Der Austausch muss also fester Teil der taktgenauen busphasenbasierten TLM-Simulation werden. Dieser Austausch von Timing-Information kann sehr gut mit Hilfe der L2-Bindungschecks bewerkstelligt werden (siehe Abschnitt 4.6.3). Dazu werden zwei spezielle Klassen definiert, die von Initiator- und Targetsockets verwendet werden müssen, um TID zu betreiben. Abbildung 4.47 zeigt diese Klassen.

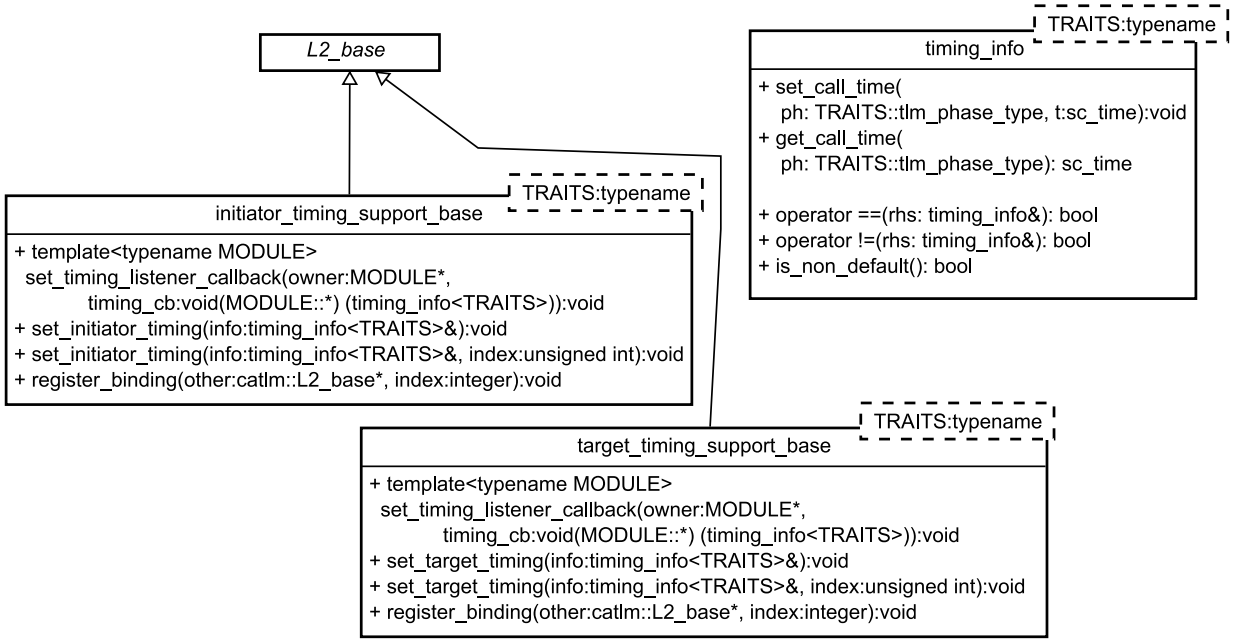


Abbildung 4.47.: Klassendiagramme für Timing-Information-Distribution-Klassen

[41] Z.B: Annahme: Ein Prozess wurde soeben durch eine Eingangskommunikation gestartet. `src` bedeutet Eingangstakt, `dst` bedeutet Ausgangstakt. Alle Variablen sind Zeitmarken. Wertet $\left(\lfloor\frac{now-start\_time_{dst}}{period_{dst}}\rfloor+1\right)*period_{dst}+start\_time_{dst} \leq (now+period_{src})$ zu *wahr* aus, so war es die letzte Flanke innerhalb des Ausgangstaktes. Ist dem nicht so, kann mit Warten um $\left(\lfloor\frac{now-start\_time_{dst}}{period_{dst}}\rfloor+1\right)*period_{dst}+start\_time_{dst}-period_{src}-now$ vom Aufrufzeitpunkt hinter die letzte Flanke des Eingangstaktes im Ausgangstakt gesprungen werden.

Da die TID mit Hilfe der L2-Bindungscheck durchgeführt wird, sind die Klassen `[inititator|target]_timing_support_base` von `L2_base` abgeleitet. Dementsprechend müssen Instanzen dieser Objekte oder davon abgeleitete Objekte mittels `set_L2_ptr` (siehe Abbildung 4.24) an die L1-Konfigurationen der Sockets übergeben werden. Der L1-Konfiguration eines Initiator-Socket muss eine `inititator_timing_support_base` und der L1-Konfiguration eines Target-Socket eine `target_timing_support_base` übergeben werden. Die Funktion `register_binding` in den jeweiligen Klassen entspricht in der Signatur einem L2-Callback (siehe wiederum Abbildung 4.24). Werden keine weiteren L2-Tests durchgeführt, können diese Funktionen direkt als L2-Callbacks bei den L1-Konfigurationen registriert werden. Wird ein eigener L2-Callback verwendet, muss `register_binding` aus diesem heraus auf der entsprechenden `timing_support_base` aufgerufen werden.

Nutzer können bei den `timing_support_base`-Klassen wiederum Callbacks registrieren (`set_timing_listener_callback`), über welchen sie dann Timing-Informationen erhalten, die von verbundenen Sockets aus übermittelt wurden. Die Übermittlung von Timing-Informationen zu verbundenen Socket geschieht mittels `set_..._timing`. Die Version ohne `index` sendet die Timing-Information zu allen verbundenen Sockets, während die Version mit `index` die Information nur an den verbundenen Socket mit entsprechendem Index übermittelt.

Die Timing-Information selbst ist ein Container, dem mittels `set_call_time` die oben erläuterte spätestmögliche Zeit nach dem Auftreten einer Taktflanke, bei der die entsprechende Phase übermittel werden kann, übergeben werden kann. Mit `get_call_time` kann diese abgefragt werden. Darüber hinaus ist es auch möglich, zwei Container miteinander zu vergleichen. Funktion `is_non_default` liefert `true`, sobald der Container eine Aufrufzeit enthält, die von Null verschieden ist.

Damit ist das Empfangen und Senden von Timing-Informationen, also die TID möglich. Die Regeln bezüglich der TID sind:

Wann muss die TID verwendet werden?

Der Nutzer muss die TID verwenden, wenn ein Modul eine Kommunikation nicht zum gleichen Simulationszeitpunkt wie das Auftreten der Taktflanke durchführt oder wenn eingehende Kommunikation noch im selben Takt die ausgehende Kommunikation auslöst und somit auch potentiell nicht zum gleichen Zeitpunkt wie die Taktflanke stattfindet.

Wann muss Timing-Informaton empfangen und gesendet werden?

Ist die Verzögerung zwischen Taktflanke und Kommunikation einzig modulintern bestimmt, z.B. weil der Nutzer innerhalb des Moduls eine partielle Ordnung der Prozesse im Modul mit Hilfe von Zeitverzögerungen realisiert, so genügt es, die entsprechende Aufrufverzögerung über die TID zu kommunizieren. Das Registrieren von Callbacks, also das Empfangen von Timing-Information, ist nicht notwendig.

Gibt es eine direkte Abhängigkeit zwischen eingehender und ausgehender Kommunikation innerhalb eines Taktes, müssen Callbacks registriert werden, damit die Startzeitpunkte der eingehenden Kommunikationen, die die ausgehende Kommunikation auslösen, bekannt sind. Gibt es keine Verzögerung zwischen der eingehenden und der ausgehenden Kommunikation, so kann die empfangene Information direkt weiter geleitet werden. Ansonsten muss die empfangene Timing-Information vor der Weiterleitung entsprechend berechnet werden.

Initialisierung der TID und Re-Evaluation

Alle Module müssen initial davon ausgehen, dass eingehende Kommunikationen zum Zeitpunkt der Taktflanke stattfinden und dies als Grundlage für die ersten zu verteilenden Informationen nutzen. Diese erste Information ist zur Elaborations-Zeit auszutauschen. Danach dürfen Timing-Informationen nur noch dann ausgetauscht werden, wenn sich bereits verteilte Informationen (aus Sicht eines Moduls) geändert haben. Empfangene Timing-Information darf nur verwendet werden, wenn sie größer als die letzte Information ist. Dies erfordert lokale Kopien der bereits verteilten und erhaltenen Informationen, und damit wird sicher gestellt, dass am Ende der Elaboration die verteilte Information stabil ist. Während der Simulation darf sich Timing-Information nur dann ändern, wenn sich Taktperioden ändern. Eine solche Änderung breitet sich dann lawinenartig durch das System aus.

Verwendung der Timing-Information und Reaktion auf Änderung zur Laufzeit

Wie oben beschrieben, können Module mit Hilfe der Timing-Information ihre Ausführung so lange verzögern, bis alle eingehenden Kommunikationen abgeschlossen sind. Während dieser Zeit kann sich aber die notwendige Verzögerung ändern (weil sich eine Taktperiode geändert hat). Folglich müssen die Module derart implementiert werden, dass die Verzögerung abgebrochen und mit der neuen Verzögerung neu gestartet werden kann. Um dies nicht unnötig zu verkomplizieren, dürfen solche Änderungen nur zum Zeitpunkt einer Taktflanke übertragen werden[42]. Damit ist sichergestellt, dass die Module noch vor Ablauf der kürzest möglichen Verzögerung (eine Femtosekunde simulierter Zeit nach der Taktflanke) die Änderung bemerken und noch keine Verzögerung stattgefunden hat, also keine komplizierten Berechnungen für das weitere Verzögern notwendig werden.

Folgerungen aus den Regeln

Die oben erläuterten Regeln erlauben es, dass vollständig synchrone Module die TID in keiner Weise benutzen müssen. Dadurch, dass sie keine Information senden, signalisieren sie, dass sie zum gleichen Zeitpunkt wie die Taktflanke kommunizieren. Da sie empfangene

[42] Bei allen von mir untersuchten Taktimplementierungen (siehe Abschnitt 4.10.2) wird eine Taktperiodenänderung, wenn überhaupt, immer erst mit einer neuen Flanke aktiv, die Anforderung kann mit diesen also eingehalten werden.

Daten erst im nächsten Takt bearbeiten, ist der genaue Empfangszeitpunkt im Takt ohne Bedeutung, und sie benötigen keine Timing-Information-Callbacks.

Die Timing-Informationen sind die spätestmöglichen Aufrufzeitpunkte, der eigentliche Aufruf kann aber auch früher geschehen.

4.12. Zusammenfassung

In diesem Kapitel wurde erläutert, wie ein TLM-Interface für die taktgenaue busphasenbasierte *J-R*-Simulation mit TLM aus einem Satz von Busphasen heraus konstruiert werden kann (siehe Abschnitt 4.2). Das konstruierte Interface basiert auf TLM-2.0, sodass ein taktgenauer busphasenbasierte Modellierungsstil für TLM-2.0 definiert wird. Dieser Modellierungsstil fußt auf den Regeln des TLM-2.0-LRM, erweitert bzw. schränkt diese ein, ändert aber nie die Semantik von Regeln oder Definitionen, sodass die bereits existierenden Modellierungsstile AT und LT nicht davon betroffen sind. Es handelt sich also um eine echte Erweiterung von TLM-2.0. Den Modellierungsstil könnte man in Anlehnung an AT und LT Cycle-Timed (CT) nennen. Das um den vorgeschlagenen CT-Modellierungsstil erweiterte TLM-2.0 im Zusammenhang mit der in Kapitel 3 definierten taktgenauen busphasenbasierten *J-R*-Simulation stellt dann meinen Vorschlag zur Standardisierung von taktgenauer TLM dar. Die Erweiterungen von TLM-2.0 im Einzelnen sind:

- Der Rückgabewert `TLM_COMPLETED` wird bei der CT-Modellierung nicht verwendet (siehe Abschnitt 4.4, Seite 49).
- Der Rückgabewert `TLM_UPDATED` wird bei der CT-Modellierung eingeschränkt und gegebenfalls zwingend verwendet (siehe Abschnitt 4.9).
- Zeit-Annotationen sind möglich, aber nicht empfohlen (siehe Abschnitt 4.4, Seite 51).
- TLM-Phasen dürfen während einer Transaktion wiederholt werden (siehe Abschnitt 4.5.1).
- Bindungschecks zur Laufzeit wurden zu TLM-2.0 als Regeln und Code hinzugefügt (siehe Abschnitte 4.5.2 und 4.6).
- Die Modifiabilities wurden klassifiziert und entsprechende Verwendungsregeln geprägt (siehe Abschnitte 4.5.2 und Abschnitt 4.7).
- Regeln zur Implementierung und Verwendung von GP-Erweiterungen, speziell für die CT-Modellierung wurden geprägt (siehe Abschnitt 4.8).
- Ein CT-Standard-Takt und all seine Interfaces wurden definiert 4.10.
- Ein Mechanismus zur partiellen Ordnung von Prozessen wurde in die CT-Modellierung übernommen (siehe Abschnitt 4.11).

Alle genannten Konzepte wurden erfolgreich implementiert [Günz10b].

5. Erweiterungen von SystemC

Inhalt

5.1. Einleitung

Bevor die praktische Anwendung der in Kapitel 4 eingeführten taktgenauen busphasenbasierten *J-R*-Simulation mit TLM-2.0 in Kapitel 6 gezeigt wird, beschäftigt sich dieses Kapitel mit der Erfüllung von Zielstellung Z3 aus Abschnitt 3.4, also mit entsprechenden Simulatorerweiterungen. Die von mir identifizierten Möglichkeiten umfassen drei verschiedene Aspekte, namentlich Event-Chains, Synchronisationsebenen und Taktimplementierung. Alle drei werden im Folgenden erläutert, und die Optimierungsmöglichkeiten werden aufgezeigt. Die Modifikationen sind nicht für die erfolgreiche taktgenaue TLM-Modellierung mit TLM-2.0 nötig. Alle bisher beschriebenen Konzepte können mit einem IEEE-1666-konformen Simulationskernel verwendet werden. Die Modifikation vereinfachen aber die Entwicklung, Wartung und Analyse der Modelle und erhöhen die Simulationsperformance.

5.2. Hinweise zu den Experimenten

Alle Experimente in dieser Arbeit wurden auf einem Intel® Pentium® 4 (Single Core) mit 3,2 GHz Taktfrequenz, 512 MB RAM, 1MB Cache, CentOS Release 5.4, gcc 4.1.2[1], OSCI SystemC-2.2.0 Release 14.03.2007[2], OSCI TLM-2.0.1 Release 15.07.2009[3] und Mentor Graphics ModelSim® SE Version 6.4a durchgeführt. Die Experimente wurden zur Minimierung von Messverfälschungen aufgrund nebenläufer Prozesse wann immer möglich im Linux Run-Level 1 (Single-User-Mode) durchgeführt. Im Fall der Messungen der Simulationszeit

[1] Maßgebliche Compiler-Option: -O3
[2] Sowohl modifiziert gemäß Kapitel 5 als auch unmodifiziert.
[3] Modifiziert gemäß Kapitel 4

des Linux-Boot-Vorgangs auf einer virtuellen Plattform (Abschnitt 5.5.4, Seite 144) wurde im Linux Run-Level 5 (Multi-User mit Netzwerk und grafischer Oberfläche) gearbeitet, da zur Darstellung des simulierten Terminals eine X11-Umgebung notwendig war.

5.3. Event-Chains

Die Simulationsperformance reaktiver Module wie Slaves oder Busmodelle kann gesteigert werden, wenn die Simulationsprozesse nicht bei jedem Takt den Zustand von durch eingehende IMC veränderbaren Variablen überprüfen, sondern wenn sie erst durch die entsprechenden IMC gestartet werden.

Listing 5.1 illustriert dies knapp an einem Beispiel. Angenommen ein Slave bearbeitet eingehende Kommunikationen taktsynchron, also bei der nächsten Taktflanke nach Empfang der Kommunikation. Dann kann er entweder bei jeder Taktflanke eine mindestens Delta-verzögerte[4] Variable abfragen (in Listing 5.1 Prozess `req_sampler`), oder er kann mit Hilfe eines Events einen Prozess starten, der dann einen Takt später sich selbst (oder einen anderen Prozess) startet, um die Kommunikationsbearbeitung vorzunehmen (in Listing 5.1 Prozess `data_sampler`).

```
1  SC_MODULE(slave){
2    sc_core::sc_port<tlm_clock_ns::tlm_clock_if> clk;
3    AHB_target_socket<slave, 32> socket;
4    SC_CTOR(slave)
5      : clk("clk") , socket("socket") , m_req_received(false)
6      , m_data_state(false) , m_exec_req_sampler(0) , m_exec_data_sampler(0)
7    {
8      socket.config_for_split_capable_bus_slave();
9      socket.register_nb_transport_fw(this, &slave::nb_transport_fw);
10     SC_METHOD(req_sampler);
11     SC_METHOD(data_sampler);
12     dont_initialize();
13     sensitive<<m_b_data_ev;
14   }
15
16   ~slave(){std::cout<<"Executed req sampler : "<<m_exec_req_sampler<<std::endl
17             <<"Executed data sampler: "<<m_exec_data_sampler<<std::endl;}
18
19   tlm::tlm_sync_enum nb_transport_fw(int, tlm::tlm_generic_payload& gp, tlm::tlm_phase& ph,
20      sc_core::sc_time& ti)
21   {
22     if (ph==tlm::BEGIN_REQ) m_req_received=true; else if (ph==BEGIN_DATA) m_b_data_ev.notify();
23     return tlm::TLM_ACCEPTED;
24   }
25
26   void req_sampler(){
27     if (m_req_received) {
28       std::cout<<sc_core::sc_time_stamp()<<" Request sampled"<<std::endl;
29       m_req_received=false;
30     }
31     m_exec_req_sampler++;
32     next_trigger(clk->edge_in_cycle(1));
```

[4] Dies vermeidet das zu frühe Erkennen einer Kommunikation, wenn der IMC im gleichen Takt aber schon vor der Ausführung des getakteten Prozesses empfangen wird.

```
33   }
34
35   void data_sampler(){
36     if (m_data_state){
37       m_data_state=false; std::cout<<sc_core::sc_time_stamp()<<" Data sampled"<<std::endl;
38     }
39     else{
40       m_data_state=true; next_trigger(clk->edge_in_cycle(1));
41     }
42     m_exec_data_sampler++;
43   }
44
45   sc_core::sc_signal<bool> m_req_received;
46   sc_core::sc_event        m_b_data_ev;
47   bool                     m_data_state;
48   unsigned int             m_exec_req_sampler, m_exec_data_sampler;
49 };
```

Listing 5.1: Code-Beispiel: Aktive und reaktive synchrone Bearbeitung eingehender IMCs

Der Slave aus Listing 5.1 wird an ein Busmodell angeschlossen und mit einem Takt mit 10 ns Periode betrieben. Die simulierte Dauer sei 1000 ns, und das Busmodell sendet bei den Flanken 10 und 21 (da die erste Flanke zum Zeitpunkt Null auftritt, also bei 90 ns bzw. 200 ns) `nb_transport_fw` mit der Phase `BEGIN_REQ` und bei den Flanken 11 und 22 (100 ns bzw. 210 ns) `nb_transport_fw` mit der Phase `BEGIN_DATA`. Dann ergibt sich eine Ausgabe wie in Listing 5.2. Man erkennt, dass beide Prozesse öfter ausgeführt werden als kommuniziert wird: `req_sampler` wird in jedem der 101 simulierten Takte gestartet, während `data_sampler` zweimal pro Kommunikation gestartet wird, da ein Start bei der eingehenden Kommunikation und ein weiterer Start für die Synchronisation mit dem Takt notwendig sind.

```
1            SystemC 2.2.0 --- Aug  5 2010 16:21:01
2        Copyright (c) 1996-2006 by all Contributors
3                    ALL RIGHTS RESERVED
4  100 ns Request sampled
5  110 ns Data sampled
6  210 ns Request sampled
7  220 ns Data sampled
8  Executed req sampler : 101
9  Executed data sampler: 4
```

Listing 5.2: Simulationsbeispiel zu aktiver und reaktiver Bearbeitung eingehender IMCs

Es wird deutlich, dass bei reaktiven Modulen, gerade bei sporadischer Kommunikation, die Ausführung zu jedem Takt zu einem erheblichen Overhead führt (im Beispiel von Listings 5.1 und 5.2 ein Overhead von 99 nutzlosen zu zwei gewünschten Ausführungen). Besser ist das reaktive Starten auf die Kommunikation hin, welche dadurch möglich ist, dass sich ein Empfänger auf den in Abschnitt 4.9 geforderten und mit Hilfe der Timing-Information-Distribution (Abschnitt 4.11) auch für kombinatorische Berechnungen sichergestellten, höchstens einmaligen, nicht wiederrufbaren Empfang einer Transaktion mit einer gewissen Phase in einem Takt verlassen kann. Jedoch zeigt Listing 5.1, dass im Falle synchroner Reaktionen[5] eine Konstruk-

[5]Kombinatorische Reaktionen können prinzipiell direkt auf die eingehende Kommunikation hin geschehen.

tion aus zwei Events (eines zum Prozess starten und eines für die eigentliche Bearbeitung) und eine Mini-Zustandsmaschine mit zwei Zuständen im Prozess notwendig wird. Dies ist fehleranfällig, nicht leicht zu warten und birgt die Gefahr von Race-Conditions, wenn der Prozess gerade auf das zweite Event wartet (in Listing 5.1 die Taktflanke), während bereits wieder eine neue Kommunikation eintrifft. Dann kann das dabei ausgelöste Event abhängig von der Ausführungsreihenfolge der Simulationsprozesse übersehen werden.

Für diese in der taktgenauen TLM-Simulation recht häufige Situation sollte eine Lösung gefunden werden, die einfacher handhabbar ist, die Mehrfachausführung minimiert (bei oben gezeigter Implementierung der synchronen reaktiven Behandlung von Kommunikation immer noch ein Overhead von zwei ungewünschten zu zwei gewünschten Ausführungen) und Race-Conditions vermeidet.

Mit den in SystemC vorhandenen Konstrukten ist es nicht möglich, die dynamische Sensitivität, also die veränderbare Liste von Events, auf die hin ein Prozess startet, von außen zu ändern. Dies kann der Prozess nur selbst, muss also gestartet werden, was zu dem oben in Listing 5.1 für den Prozess `data_sampler` gezeigten Konstrukt führt. Es bleibt also eine möglichst einfache Modifikation des Simulation-Kernels zu finden, die unter keinen Umständen die Ausführungssemantik des Kernels gefährdet.

5.3.1. Lösungsansatz

Die von mir für diese Problem vorgeschlagene Lösung nenne ich **Event-Chain**. Eine Event-Chain ist eine Verbindung mindestens zweier Events. Dabei ist ein Event das **Ur-Event**, das andere das **Folge-Event**. Ein Event kann Ur-Event mehrerer verschiedener Folge-Events sein. Ein Folge-Event kann wiederum Ur-Event eines weiteren Folge-Events sein. Die folgenden Regeln gelten für Event-Chains:

Ur-Event :

1. Der Aufbau und Abbau der Event-Chain erfolgt über das Ur-Event.
2. Eine augenblickliche Auslösung (`notify()`) des Ur-Events, führt zu einer augenblicklichen Auslösung aller verbundenen Folge-Events.
3. Eine Deltaschritt-verzögerte Auslösung (`notify(SC_ZERO_TIME)`) des Ur-Events führt zu einer unmittelbaren Auslösung aller nach Ablauf der Verzögerung noch verbundenen Folge-Events.
4. Eine zeitverzögerte Auslösung (`notify(sc_time)`) des Ur-Events führt zu einer unmittelbaren Auslösung aller nach Ablauf der Verzögerung noch verbundenen Folge-Events.
5. Ein Abbruch von Deltaschritt- oder zeitverzögerten Auslösungen (`cancel()`) des Ur-Events unterbindet auch die entsprechenden Auslösungen der Folge-Events (Regeln 3 und 4). Die Verkettung bleibt aber erhalten.
6. Eine Event-Chain ist entweder eine **einmalige** oder eine **permanente Verkettung**.

6.1 Eine einmalige Verkettung wird nach einer Auslösung des Folge-Events automatisch abgebaut.

6.2 Eine permanente Verkettung wird nach einer Auslösung des Folge-Events nicht abgebaut, kann aber vom Ur-Event explizit abgebaut werden.

Folge-Event :

1. Ein Folge-Event ist sich seiner Verkettung nicht gewahr[6]. Eine entsprechende Auslösung ist für das Event nicht unterscheidbar von einem direkten Aufruf von `notify()`.

1.1 Folglich hat `cancel()` auf einem Folge-Event keinen Einfluss auf durch Deltaschritt- oder zeitverzögerte Auslösungen des Ur-Events verursachte Auslösungen.

Die Möglichkeit zur Erzeugung einer Event-Chain wurde wie in Listing 5.3 direkt in den Code für das `sc_event` integriert. Die dort gezeigten Funktionen haben, wenn sie auf einer Instanz `ev` eines Events aufgerufen werden, folgende Effekte: Die Funktion `chain_permanent` verkettet das übergebene Event `other` permanent als Folge-Event mit dem Ur-Event `ev`. Die Funktion `unchain_permanent` hebt eine permanente Verkettung zwischen Ur-Event `ev` und Folge-Event `other` auf. Die Funktion `chain_once` verkettet das übergebene Event `other` einmalig als Folge-Event mit Ur-Event `ev`. Rückgabewert ist dabei ein Schlüssel, der zur Aufhebung der Verkettung notwendig ist[7]. Die Funktion `unchain_permanent` hebt eine permanente Verkettung von Ur-Event `ev` mit Folge-Event `other` auf. Funktion `unchain_all` hebt alle Verkettungen des Ur-Events `ev` auf.

```
1  class sc_event{
2    ...
3  public:
4    void chain_permanent(sc_event& other);
5    void unchain_permanent(sc_event& other);
6    unsigned int chain_once(sc_event& other);
7    void unchain_once(unsigned int);
8    void unchain_all();
9    ...
10 };
```

Listing 5.3: Modifikation der Klasse `sc_event`

5.3.2. Funktionsweise

Mit Hilfe der Funktionen aus Listing 5.3 hat ein (Ur-)Event stets einen Datensatz an aktuell verketteten Folge-Events. Wird ein Event ausgelöst (direkt durch `notify()` oder nach Ablauf einer Deltaschritt- oder Zeitverzögerung vom Simulationskernel), so kann es einfach auf all diesen Events wiederum `notify()` auslösen. Danach wird der Datensatz der einmalig verketteten Events gelöscht.

[6] Es gibt für ein Event keine Möglichkeit festzustellen, ob es selbst ein Folge-Event ist. Nur ein Ur-Event kennt die Folge-Events.

[7] Aufhebungen von einmaligen Verkettungen sind in meinen Experimenten signifikant häufiger notwendig geworden als Aufhebungen permanenter Verkettungen. Zu Gunsten der Simulationsperformance wird durch die Verwendung eines Aufhebungsschlüssels ein zeitaufwändiges Suchen in der Datenstruktur der einmalig verketteten Folge-Events vermieden.

5.3.3. Fazit

Event-Chains erlauben die Implementierung von reaktiven taktsynchronen Bearbeitungen von Kommunikationen wie in Listing 5.4 gezeigt (vergleiche dazu die gleichlautenden Funktionen in Listing 5.1). Der Prozess `data_sampler` ist nach wie vor statisch sensitiv auf `m_b_data_ev`. Wird eine Kommunikation empfangen, wird dieses Event einmalig mit dem nächsten Takt-Event verkettet, sodass `data_sampler` genau einmal zum gewünschten Zeitpunkt pro Kommunikation ausgeführt wird.

Der Code wird deutlich vereinfacht und Mehrfachausführungen werden eleminiert. Auch Race-Conditions zwischen einem erneuten eingehenden IMC und einer gleichzeitigen Ausführung des Prozesses werden aufgelöst, da ein neuer IMC eine neue Verkettung zwischen dem nächsten Takt-Event und dem statischen Event aufbaut, die im nächsten Takt sicher ausgelöst und so den Prozess starten wird.

```
1  tlm::tlm_sync_enum nb_transport_fw(int, tlm::tlm_generic_payload& gp, tlm::tlm_phase& ph, sc_core::sc_time& ti)
2  {
3    if (ph==tlm::BEGIN_REQ) m_req_received=true;
4    else if (ph==BEGIN_DATA) clk->edge_in_cycle(1).chain_once(m_b_data_ev);
5    return tlm::TLM_ACCEPTED;
6  }
7
8  void data_sampler()
9  {
10   std::cout<<sc_core::sc_time_stamp()<<" Data sampled"<<std::endl;
11   m_exec_data_sampler++;
12 }
```

Listing 5.4: Code-Beispiel zu reaktiver synchroner Bearbeitung von IMCs mit Event-Chains

Zur Untersuchung der Simulationsperformance wurde die in Listings 5.1 und 5.4 illustrierten Techniken zur taktsynchronen Bearbeitung eingehender IMC unabhängig voneinander vermessen. Ein in allen Fällen identischer Traffic-Generator erzeugt in einem gewissen Prozentsatz aller simulierter Takte `nb_transport`-Aufrufe, die vom Slave taktsynchron bearbeitet werden sollen. Diese Bearbeitung besteht darin, lediglich die Anzahl empfangener IMC zu zählen, ansonsten findet keine weitere Aktivität statt. Takte, in denen kein `nb_transport`-Aufruf stattfinden soll, werden vom Traffic-Generator mit `edge_in_cycle` aus dem TLM-Takt-Interface (siehe Abschnitt 4.10.3 und Abbildung 4.42) übersprungen. Es werden 20 Millionen Takte vom Traffic-Generator simuliert, danach beendet er die Simulation.

Abbildung 5.5 zeigt die Ergebnisse der Messung bei steigender Anzahl von Takten, in denen `nb_transport` aufgerufen wird. Ein in jedem Takt aktiver Prozess (Messreihe „aktives Sampling“) dominiert die Simulationsperformance aufgrund seiner Aktivierung in jedem Takt. Es existieren keine Takte, die vollkommen irrelevant sind und somit übersprungen werden können. Die Aktivierungskosten des Prozesses sind gegenüber der trivialen Aktivität im Prozess so hoch, dass keine messbare Abhängigkeit zu den tatsächlich aktiven Takten besteht.

Die reaktive taktsynchrone Bearbeitung mit dem für den `data_sampler`-Prozess in Listing 5.1 illustrierten Konstrukt aus zwei Events und Mini-Zustandsautomat (Messreihe „2-Event-

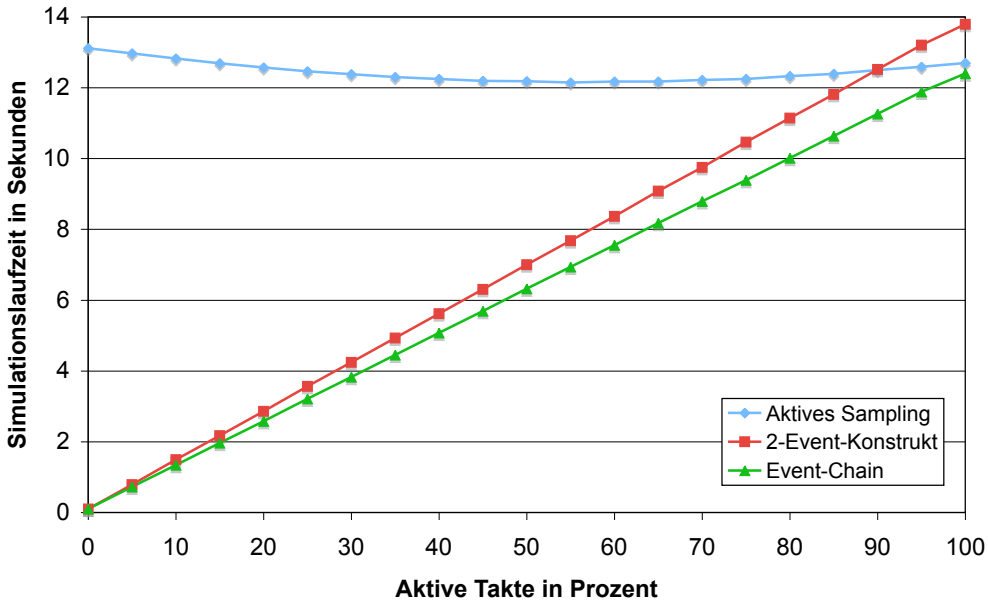


Abbildung 5.5.: Simulationslaufzeiten verschiedener Techniken zur taktsynchronen Bearbeitung eingehender IMC in Abhängigkeit von der Kommunikationshäufigkeit

Konstrukt“) kann erwartungsgemäß stark von ungenutzten Takten profitieren. Bei einer hohen Anzahl von aktiven Takten ist die Performance aber schlechter als bei einem stets aktiven Takt. Dies begründet sich durch die Mehrfachaktivierungen, da für jeden aktiven Takt zwei Prozessausführungen stattfinden und dann der Prozess deutlich öfter ausgeführt wird, als es überhaupt simulierte Takte gibt[8].

Die Event-Chain-Lösung profitiert ebenso von ungenutzten Takten und erreicht im schlimmsten Fall die Laufzeit des stetig aktiven Prozesses. Es handelt es sich bei der Event-Chain somit um einen Ansatz, der bei sporadischer Kommunikation den stetig aktiven Prozessen aufgrund seiner Performance und dem 2-Event-Konstrukt aufgrund seiner einfacheren Handhabung und der Vermeidung von Race-Conditions deutlich vorzuziehen ist. Da selbst im schlimmsten Fall (100 Prozent aktive Takte) die Performance mit der eines stetig aktiven Prozesses vergleichbar ist, sollte für die taktsynchrone reaktive Bearbeitung von eingehenden IMC immer eine Event-Chain verwendet werden.

5.4. Synchronisationsebenen

Die in Abschnitt 4.11 eingeführte Timing-Information-Distribution (TID) ermöglicht es dem Nutzer, die Ausführungsreihenfolge von Prozessen innerhalb eines Taktes partiell zu ordnen. Dadurch kann sichergestellt werden, dass kombinatorische Berechnungen nur exakt einmal ausgeführt werden. Wie in [GüKA07] belegt, bietet die TID die bestmögliche Simulationsperformance bei minimalem Code-Overhead und kann mit jedem IEEE1666-konformen SystemC-Simulationkernel verwendet werden. Der einzige, aber gravierende Nachteil ist, dass

[8] Jedoch ist eine Einzelaktivierung günstiger als die des stetig aktiven Prozesses, da nicht teuer auf ein `sc_signal` zugegriffen werden muss und jedes zweite Mal die performantere statische Sensitivität greift.

durch die Verwendung der TID zur Ordnungserzeugung Zeitverzögerungen eingesetzt werden. Somit entstehen Zeitartefakte in den Simulationsoutputs, die keinerlei Relation zur RTL-Referenz haben. Dies erschwert automatisierte Vergleiche zwischen TLM- und RTL-Simulation, die gerade während der Entwicklung taktgenauer TLM-Modelle wichtig sind. Auch werden die erzeugten Outputs für den Nutzer schlechter lesbar.

Ideal wäre eine im Simulationskernel integrierte Möglichkeit zur partiellen Prozessordnung, die ohne Modifikationen der Simulationszeit arbeitet. Es ist wichtig, dass dieser Mechanismus die TID aber nicht obsolet machen wird. Für kombinatorische Berechnungen zwischen Takt-Domänen wird weiterhin die Information benötigt, wann der spätest mögliche Zeitpunkt innerhalb eines Taktes ist, an dem eine Kommunikation auftreten kann. Diese Zeiten sind aber auch in einer RTL-Simulation beobachtbar. Geht zum Beispiel ein kombinatorischer Pfad von einer 100 MHz Takt-Domäne in eine gleichphasige 50 MHz Takt-Domäne, so ist der spätest-mögliche Zeitpunkt, zu dem eine 100 MHz Takt-Domänenkommunikation bei der 50 MHz Takt-Domäne eingehen kann, 5 ns nach der 50 MHz Taktflanke. Offenbar ist dies auch in RTL so, und es handelt sich nicht um ein Artefakt. Die im Folgenden vorgeschlagene Möglichkeit zur Erzeugung partieller Ausführungsordnungen erlaubt dann die Ordnung innerhalb eines Zeitschrittes.

5.4.1. Lösungsansatz

Ziel ist, innerhalb eines Zeitschrittes eine partielle Prozess-Ausführungsordnung (PPAO) zu ermöglichen. Die einzige Ordnung innerhalb eines Zeitschrittes, die bereits im Kernel existiert, ist die der Deltazyklen. Innerhalb eines Deltazyklus gibt es keine Ausführungsordnung. Wie in [GüKA07] gezeigt, können aber Deltazyklen nicht effizient für die PPAO verwendet werden, da man nicht auf einen bestimmten Deltazyklus warten kann, sondern immer nur auf den nächsten. Dies liegt in der Natur der Deltazyklen, da diese nur dynamisch entstehen, wenn jemand eine Deltaverzögerung fordert. Wollte man mit Hilfe von Deltazyklen eine PPAO aufbauen, müsste im Zweifelsfalle in einer Schleife die notwendige Anzahl von Deltazyklen abgewartet werden. Dies führt zu vielen Kontextwechseln und somit zu Performanceeinbußen.

Es muss also eine zustäzliche Art der Ordnung geschaffen werden. Die von mir vorgeschlagene Lösung nenne ich **Synchronisationsebenen**. Die Synchronisationsebene, auf der getaktete Prozesse ausgeführt werden, nenne ich Synchronisationsebene 0. Jedes Mal, wenn ein neuer Simulationszeitpunkt beginnt, wird die Synchronisationsebene auf 0 gesetzt. Die Synchronisationsebene sl, auf der eine kombinatorische Berechnung ausgeführt werden muss, ergibt sich zu $sl = \max(sl_{in1}, sl_{in2}, ...)+1$, wobei $sl_{in1}, ...$ die Synchronisationsebenen der Prozesse sind, die die Eingangsdaten für die Berechnung liefern.

Die Kenntnis, auf welcher Synchronisationsebene ein Prozess innerhalb eines Moduls auszuführen ist, ergibt sich aus den kombinatorischen Abhängigkeiten innerhalb des Moduls und der Synchronsiationsebenen eingehender Kommunikation. Die modulinternen Abhängigkeiten sind dem Modulentwickler bekannt, die Synchronisationsebenen eingehender Kommuni-

kation aber nicht. Dazu wird eine Informationsdistribution ähnlich zur TID benötigt. Genau genommen kann die Synchronisationsebenen-Information über die TID betrieben werden, indem die Klasse `timing_info` aus Abbildung 4.47 (Seite 113) entsprechend Abbildung 5.6 erweitert wird (Erweiterungen sind grau hinterlegt). Damit kann dann die TID sowohl dafür verwendet werden, echte Zeitversätze zu signalisieren und auch die Synchronisationsebenen zu übertragen. Die dabei zu beachtenden Regeln stimmen mit denen für die TID (Abschnitt 4.11 Seiten 114ff) überein[9].

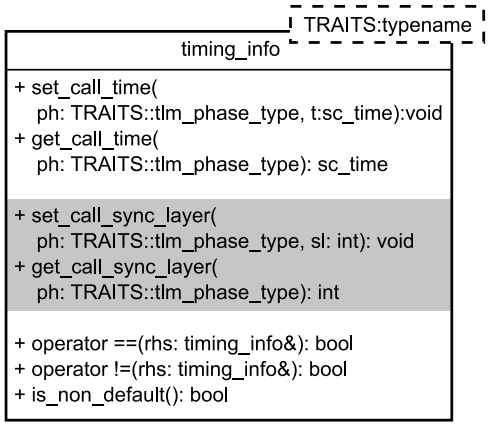


Abbildung 5.6.: Erweiterte Timing-Info-Klasse

Zur Verdeutlichung soll ein einfaches Beispiel dienen. Ein kombinatorischer Arbiter leitet innerhalb eines Taktes von höchstens zwei als gültig markierten Eingangssignalen eines zum Ausgang weiter und setzt für dieses weitergeleitete Signal eine Bestätigung. Im darauf folgenden Takt wird erneut arbitriert. Im TLM-Modell seien das Setzen eines gültigen Eingangssignales zum Arbiter mit `BEGIN_REQ` und die Bestätigung mit `END_REQ` modelliert. Dann musss der Arbiter innerhalb eines Taktes eines von maximal zwei GPs zum Ausgang weiter leiten und für dieses dann ein `END_REQ` senden. Hierzu muss er erst sicher sein, dass die sendenden Prozesse ihre GPs auch gesendet haben. Der Prozess im Arbiter muss also nach den Prozessen in den Sendern ausgeführt werden, es muss eine PPAO aufgebaut werden. Unter der Annahme, dass es nur eine Takt-Domäne gibt, zeigt Abbildung 5.7 das Ergebnis der Synchronisationsebenen-Informations-Distribution. Die Master in Abbildung 5.7 sind alle getaktet und senden somit auf Synchronisationsebene 0. Ein Arbiter muss wie oben beschrieben immer eine Synchronisationsebene später ausgeführt werden als die späteste Synchronisationsebene seiner Eingangskommunikationen.

[9]Eine etwas elegantere, halbautomatische Verteilung der Synchronisationsebenen-Information wird in [Melz09] beschrieben.

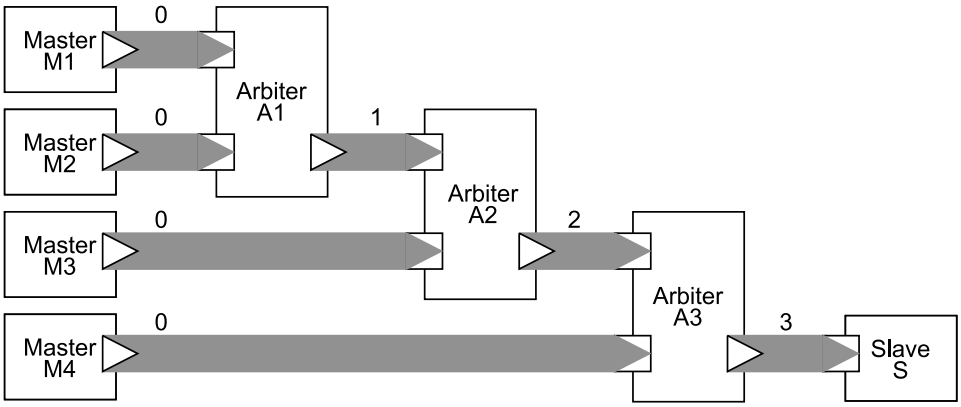


Abbildung 5.7.: Synchronisationsebenen für die Übermittlung von `BEGIN_REQ` in einem simplen Beispiel

```
1 namespcae sc_core{
2   unsigned int sls_get_current_sl();
3   void sls_request_update(unsigned int sl);
4   void sls_request_update(unsigned int sl, sc_process_handle proc);
5 }
```

Listing 5.8: Zusätzliche SystemC-Kernelfunktionen zur Verwendung von Synchronisationsebenen für die partielle Prozess-Ausführungsordnung

Damit diese Information auch für die PPAO verwendet werden kann, wird SystemC um neue Funktionen erweitert, die in Listing 5.8 gezeigt sind. Die Funktion `sls_get_current_sl`[10] gibt die aktuelle aktive Synchronisationsebene zurück. Funktion `sls_request_update` in Zeile 3 von Listing 5.8 fordert den Simulationskernel auf, die SC_METHOD, die die Funktion aufgerufen hat, auf Synchronisationsebene `sl` erneut zu starten. Die SC_METHOD verliert dadurch nicht die Kontrolle, sondern muss diese danach selbsttätig aufgeben[11]. Funktion `sls_request_update` in Zeile 4 von Listing 5.8 fordert den Kernel auf, die als Argument `proc` übergebene SC_METHOD auf Synchronisationsebene `sl` erneut zu starten[12].

Mit Hilfe dieser Funktionen kann die kombinatorische Arbitrierung aus dem Beispiel in Abbildung 5.7 sehr elegant wie in Listing 5.9 implementiert werden. Eine eingehende Kommunikation speichert das empfangene GP und fordert dann einen Start der SC_METHOD `arbitrate` auf der entsprechenden Synchronisationsebene an. Durch die vorhergehende Informationssdistribution ist sichergestellt, dass eingehende Kommunikation stets auf einer niedrigeren Synchronisationsebene als `m_sl` stattfindet.

Startet dann die SC_METHOD, testet sie nochmals die Synchronisationsebene. Danach wird sie, wenn vorhanden, das GP von Index 0 weiterleiten (in diesem Beispiel hat es Priorität), eine Bestätigung senden und dann das GP vergessen. War gleichzeitig eine GP an Index 1 vorhanden, wird die SC_METHOD im nächsten Takt erneut ausgeführt, um sicherzustel-

[10] `sls` steht hier für Synchronization Layer Support.
[11] Dies entspricht dem Verhalten von `next_trigger`.
[12] Es werden nur SC_METHODs unterstützt, da eine Unterstützung von SC_THREADs derart möglich ist, dass ein SC_THREAD auf ein Event wartet, welches von einer SC_METHOD ausgelöst wird, deren Start für eine bestimmte Synchronisationsebene geplant ist.

len, dass das noch nicht bestätigte GP von Index 1 bearbeitet wird. Bei einer Bearbeitung von Index 1, wenn also kein GP an Index 0 vorhanden war, ist kein Neustart nötig. Es kann dann wieder auf eingehende Kommunikation gewartet werden.

```
/* Erlaeuterungen:
unsigned int m_sl ist die mit Hilfe der Informationsdistribution bestimmte Synchronisationseben
fuer die Arbitrierungsentscheidung.
Die Klasse Arbiter verwendet einen Multi-Socket mit nur einem nb_transport-Callback.
Der Aufrufer wird ueber das Argument 'index' identifiziert.
tlm_generic_payload* m_payloads[2] ist ein Member der Klasse Arbiter, der fuer jeden
Sender ein Payloadzeiger speichern kann. Ist der Zeiger NULL ist kein gueltiges
Payload vorhanden.
bool m_armed ist ein Member der Arbiter-Klasse.
sc_process_handle m_proc ist Handle auf SC_METHOD 'arbitrate'
*/

//Die SC_METHOD der Arbiter-Klasse
void Arbiter::arbitrate()
{
  if(sc_core::sls_get_current_sl()<m_sl){ //wenn vom Takt gestartet ist dies immer 'true'
    if (!m_armed){ //sicherstellen, dass nur ein request_update pro Zeitschritt passiert
      //kommt entweder durch edge_in_cycle (siehe unten) oder nb_transport
      m_armed=true;
      sc_core::sls_request_update(m_sl);
    }
    return;
  }
  m_armed=false;
  tlm_phase ph(tlm::BEGIN_REQ);
  sc_core::sc_time ti;
  if (m_payloads[0]){ //index 0 hat Prioritaet
    output_socket->nb_transport_fw(*m_payloads[0], ph, ti);
    ph=tlm::END_REQ;
    input_socket[0]->nb_transport_bw(*m_payloads[0], ph, ti);
    m_payloads[0]=NULL;
    if (m_payloads[1]) next_trigger(clk->edge_in_cycle(1)); //im naechsten Takt ist was zu tun
  }
  else
  if (m_payloads[1]){
    output_socket->nb_transport_fw(*m_payloads[1], ph, ti);
    ph=tlm::END_REQ;
    input_socket[1]->nb_transport_bw(*m_payloads[1], ph, ti);
    m_payloads[1]=NULL;
  }
}

tlm_sync_enum nb_transport_fw(int index, tlm_generic_payload& gp, tlm_phase& ph, sc_core::sc_time& ti)
{
  if(sc_core::sls_get_current_sl()<m_sl){
    if (!m_armed){
      m_armed=true;
      sc_core::sls_request_update(m_sl, m_proc);
    }
  }
  else assert(0 && "das kann nicht passieren!");
  m_payloads[index]=&gp;
  return TLM_ACCEPTED;
}
```

Listing 5.9: Kombinatorische Arbitrierung mit Hilfe von Synchronisationsebenen

5.4.2. Funktionsweise

Die Unterstützung von Synchronisationsebenen wird mit einer Änderung des Simulationskernels erwirkt. Abbildung 5.10 zeigt diese Änderung (vergleiche dazu die unveränderte Arbeitsweise in Abbildung 2.4 auf Seite 11). Der Kernel enthält nun neben den Listen der Deltaschritt- und zeitverzögerten Eventauslösungen noch eine geordnete Liste mit den für verschiedene Synchronisationsebenen zurückgestellten Prozessen. Im Zustand Initialisierung wird die globale Synchronisationsebene (abbrufbar über `sls_get_current_sl`) auf Null gesetzt. Danach werden regulär beliebig viele Deltazyklen simuliert. Alle während dieser Zeit mittels `sls_request_update` zurückgestellten Prozesse werden nach den angeforderten Synchronisationsebenen geordnet in die Liste mit den für verschiedene Synchronisationsebenen zurückgestellten Prozessen einsortiert. Ist kein Prozess mehr aufgrund von Deltaschritt-Verzögerungen ausführbar, wird die Synchronisationsebene auf die nächste Ebene erhöht, für die Prozesse zurückgestellt wurden. Die globale Synchronisationsebene wird auf diesen Wert gesetzt und die Prozesse werden ausführbar gemacht. Danach werden wieder beliebig viele Deltazyklen auf dieser Synchonisationsebene ausgeführt. Nur wenn es keine für höhere Synchronisationsebenen zurückgestellten Prozesse mehr gibt, werden die simulierte Zeit erhöht, die globale Synchronisationsebene wieder auf Null gesetzt und die zeitverzögerten Ereignisauslösungen werden ausgeführt. Man beachte, dass es als Fehler angesehen wird, wenn man auf einer Synchronisationsebene X die Ausführung auf Ebene Y mit $Y \leq X$ anfordert.

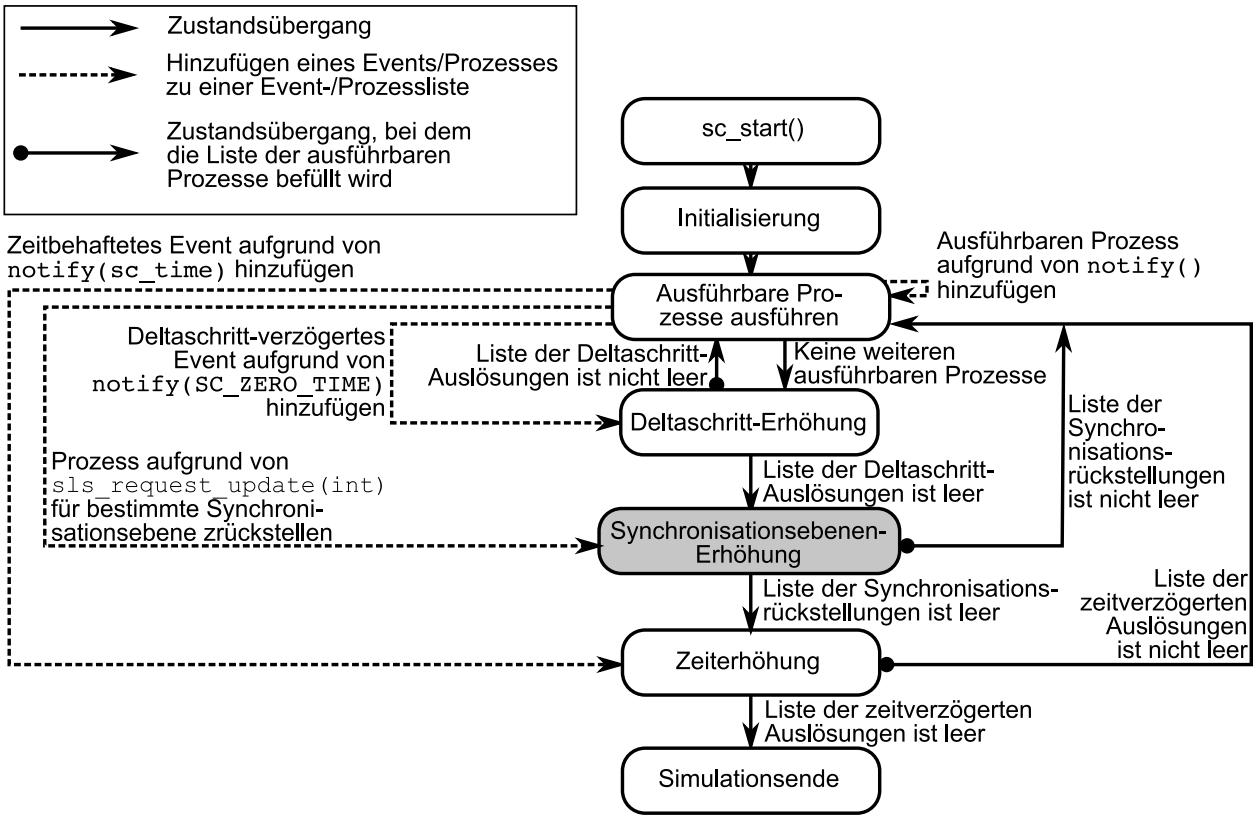


Abbildung 5.10.: Arbeitsweise des um Synchronisationsebenen erweiterten Simulators

5.4.3. Fazit

Die partielle Prozess-Ausführungsordnung (PPAO) mit Hilfe von Synchronisationsebenen vermeidet unerwünschte Simulationszeitartefakte, wie sie bei der PPAO mit Hilfe von Zeitverzögerungen entstehen. Dies erleichtert die Analyse und Auswertung von Simulationen und Modellen. Dabei bleibt der Code-Overhead beherrschbar, und die Simulations-Performance ist mit der von Zeitverzögerungen vergleichbar, da in beiden Fällen die Verwaltung von geordneten Listen den Gesamtrechenaufwand dominiert. Die Änderungen am SystemC-Kernel sind minimal und habe keine messbare Auswirkung, wenn keine PPAO verwendet werden soll. Diese Aussagen werden von Messungen aus [GüKA07] gestützt.

In Abbildung 5.11 werden die verschiedenen in [GüKA07] untersuchten Möglichkeiten der PPAO gegenübergestellt. Die verschiedenen Ansätze werden in der folgenden Liste kurz erklärt und in [GüKA07] näher erläutert.

Deltazyklen : Dieser Ansatz verwendet eine Informations-Distribution ähnlich wie die TID, damit klar ist, in welchem Deltazyklus nach der Taktflanke eine Kommunikation spätestens auftritt. Diese Information wird dann genutzt, um die Prozessausführung in den richtigen Deltazyklus zu verschieben.

Zeitverzögerung : Dies ist der auf der TID basierende, in Abschnitt 4.11 erläuterte Ansatz, der Zeitverzögerungen einsetzt, um die Prozessausführungen zu ordnen.

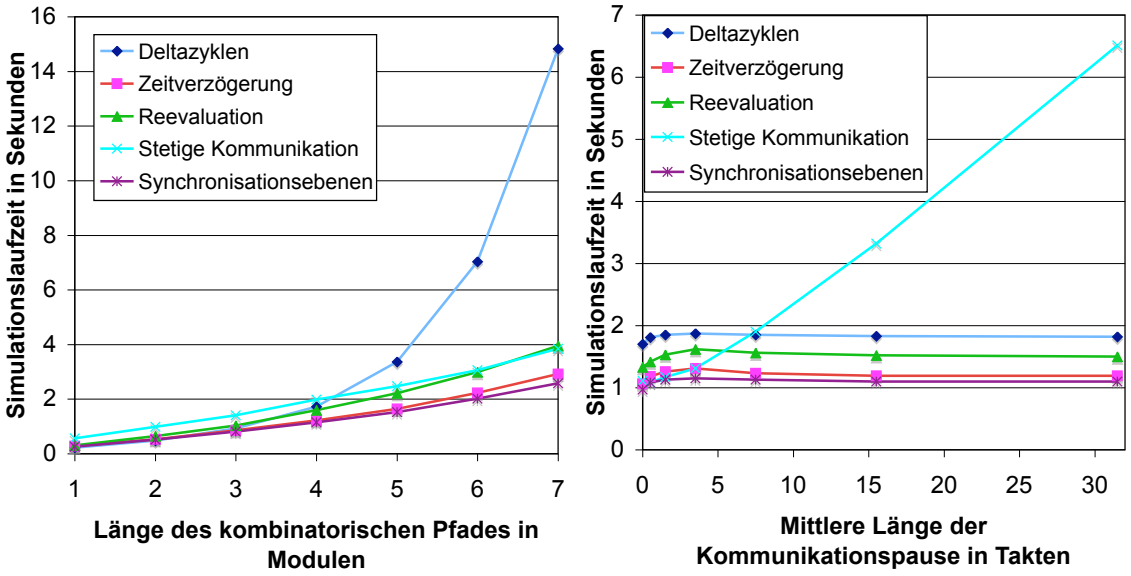
Reevaluation : Dieser Ansatz erlaubt mehrfache Kommunikationen innerhalb eines Zeitschritts, wobei auch „falsche“ Zwischen-Kommunikationen auftreten können. Die letzte Kommunikation innerhalb eines Zeitschrittes ist dann die endgültige. Dies ist vergleichbar mit der Arbeitsweise von RTL-Simulationen.

Stetige Kommunikation : Dabei wird in jedem Takt kommuniziert. Soll keine reale Kommunikation stattfinden, wird eine spezielle, leere Kommunikation durchgeführt. Dadurch können kombinatorische Prozesse immer auf alle eingehenden Kommunikationen warten, bevor sie selbst kommunizieren.

Synchronisationsebenen : Dies ist der in diesem Abschnitt beschriebene Ansatz.

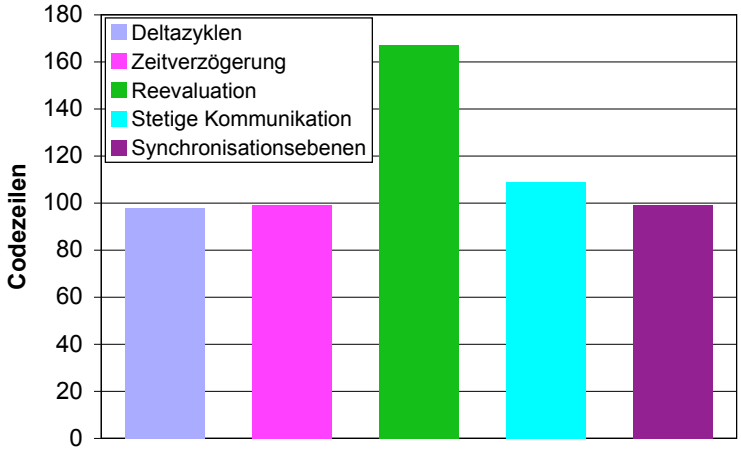
Das zugrundeliegende Experiment entspricht dem Aufbau aus Abbildung 5.7. Die variablen Parameter im Experiment sind die Anzahl der kaskadierten Arbiter, um die Länge des kombinatorisch durchlaufenen Pfades zu modifizieren und die mittlere Pausenlänge zwischen zwei Kommunikationen eines Masters. Darüber hinaus wurde untersucht, wieviel Zeilen Code für die Verhaltensmodellierung des Arbiters notwendig sind[13].

[13]Dabei wurde konsequent der in [KeRi88] verwendete Einrückungsstil, auch bekannt als 1TBS- oder K&R-Stil verwendet und die Zeilenzählung nach Anwendung des C++-Präprozessors und Löschung von Leerzeilen durchgeführt.



(a) Simulationslaufzeit in Abhängigkeit von der Länge des kombinatorischen Pfades

(b) Simulationslaufzeit in Abhängigkeit von der mittleren Länge der Kommunikationspause



(c) Zur Verhaltensmodellierung benötigte Codezeilen in Abhängigkeit vom verwendeten Synchronisationsschema

Abbildung 5.11.: Ergebnisse zu den Untersuchungen zu Synchronisationsebenen

Man erkennt in Abbildung 5.11a das bereits oben erläuterte erwartete Einbrechen der Simulationsperformance mit steigender Länge des kombinatorischen Pfades[14] bei der Verwendung von Deltazyklen zur PPAO. Abbildung 5.11b illustriert, wie mit zunehmend seltenerer Kommunikation[15], also längeren mittleren Pausen zwischen Kommunikationen, die Performance der stetigen Kommunikation einbricht. Grund dafür ist, dass die anderen Ansätze, die in diesen Pausen gar nichts tun müssen, von der Länge der Pause prinzipiell unabhängig sind, während mit steigender Pausenlänge die Anzahl der leeren Kommunikationen bei der

[14]Dabei betrug die mittlere Länge der Kommunikationspause 7,5 Takte.
[15]Dabei betrug die Länge des kombinatorischen Pfades 4..

stetigen Kommunikation steigt. In Abbildung 5.11c wird deutlich, dass für die Reevaluation erheblich mehr Code benötigt wird als für die anderen Ansätze, da Code notwendig ist, der falsche Kommunikationen rückgängig machen kann. Die stetige Kommunikation benötigt ein wenig mehr Code als die Ansätze mit Deltazyklen oder Zeitverzögerungen, da die leeren Kommunikationen behandelt werden müssen. Der marginale Mehraufwand für die Synchronisationsebenen sind in Listing 5.9 die Abfragen der aktuellen Synchronisationsebene und das Setzen der Boolschen Variablen zum Schutz gegen Mehrfachausführungen.

Wie erwartet sind die einzigen Ansätze, deren Performance in jedem Fall gut ist und mit wenig Code-Overhead auskommen, die Verwendung von Zeitverzögerungen und Synchronisationsebenen. Da Synchronisationsebenen die bereits erwähnten Simulationszeitartefakte unterdrücken, sollten zur PPAO innerhalb eines Zeitschrittes wenn möglich Synchronisationsebenen verwendet werden. In den Experimenten wurde die Unterstützung für Synchronisationsebenen derart implementiert, dass bei Nicht-Verwendung von Synchronisationsebenen keinerlei Overhead im Kernel erzeugt wird.

Es soll nochmals darauf hingewiesen werden, dass die TID aber nach wie vor bei kombinatorischen Pfaden über Taktdomänen-Grenzen hinweg nötig ist (siehe Einführung dieses Abschnitts auf Seite 124).

5.5. Taktimplementierung

Wie in Abschnitt 3.2 erläutert kann die taktgenaue J-R-Simulation mit TLM-2.0 effizienter als die zugrunde liegende RTL-Simulation ausgeführt werden, wenn es viele Takte gibt, in denen keine J-konformen Input- bzw. gemäß R relevanten Output-Änderungen auftreten. Diese Änderungen müssen nicht berücksichtigt oder berechnet werden, und Takte, in denen dies für alle betrachteten Signale gilt, können prinzipiell übersprungen werden. Dies sind in der Regel Takte, in denen entweder gar nicht kommuniziert wird oder aber in denen Busphasen mit langer Dauer stattfinden. Aus diesem Grund ist es wichtig, dass das Überspringen von Takten so effizient wie nur irgend möglich behandelt wird. Wie in Abschnitt 4.10.2 erläutert, sind mir drei verschiedene Möglichkeiten zur Taktmodellierung bekannt: Taktung mit `sc_clock`, Taktung mit zeitbehaftetem Warten und die Taktung nach Grellier. Wie in [Grel08] gezeigt, ist die Taktung nach Grellier im Worst-Case (kein Takt wird übersprungen) so effizient wie eine `sc_clock` und ansonsten effizienter. Als echte Alternativen verbleiben also nur noch die Taktung nach Grellier und das zeitbehaftete Warten. Wie in Abschnitt 4.10.2 beschrieben, wartet bei der Taktung mit zeitbehaftetem Warten jedes Modul selbsttätig die zum Überspringen der Takte notwendige Zeit ab. Module benötigen lediglich Informationen über die Taktperiode, um diese Art der Taktung zu verwenden[16].

[16] Die Unterstützung von Starts, Stopps und Frequenzänderungen mit dieser Art der Taktung ist äußerst kompliziert; sowohl in der Implementierung als auch in der Anwendung. Details dazu würden aber den Rahmen der Arbeit sprengen.

Mit Hilfe des in Abschnitt 4.10.3 eingeführten Taktinterfaces ist eine Taktimplementierung in der Lage, den nächsten zwingend zu simulierenden Takt zu identifizieren, da Prozesse über `edge_in_cycle` explizit benennen müssen, welche Taktflanke, gemessen von der aktuellen Flanke, sie als nächstes benötigen. Grellier hat dies in [Grel08] derart umgesetzt, dass die Taktimplementierung einen Ringpuffer von n Event-Zeigern enthält, sodass für das Argument t von `edge_in_cycle(t)` $t \leq n$ gilt[17]. Die Taktimplementierung enthält einen Modulo-n-Zähler, der den Index des aktuellen Elements des Ringpuffers (die aktuelle, potentielle Flanke) repräsentiert. Dazu existiert ein Prozess, der, wenn der Takt nicht gestoppt ist, in Abständen der Taktperiode reaktiviert wird, den Modulo-n-Zähler inkrementiert und das Event, auf das der Event-Zeiger im entsprechenden Ringpufferelement zeigt, auslöst und danach in einen Pool zurücklegt. Ist der Zeiger Null, wird nichts ausgelöst. Fordert ein Prozess mittels `edge_in_cycle(t)` ein Flanken-Event ab und ist der Wert des Modulo-n-Zählers m, so wird Ringpufferelement $((t+m) \mod n)$ auf Null überprüft. Ist es Null, wird ein Event aus einem Pool entnommen, und der Zeiger wird auf dieses Event gesetzt, und das Event wird zurückgegeben. Ist der Zeiger nicht Null, wird er direkt zurückgegeben.

Damit werden die abgeforderten Ereignisse immer korrekt ausgelöst, und das Abzählen nicht notwendiger Takte geschieht nur in einem einzigen Prozess (dem in der Taktimplementierung) anstatt in jedem getakteten Prozess, wie es der Fall ist, wenn man immer nur das nächste Taktevent abfordern kann.

Auch von Grellier selbst wurde die Problematik identifiziert, dass durch diese Art der Implementierung eine untere Schranke für die Simulationslaufzeit eines beliebigen Modells mit fester Simulationsdauer eingeführt wird: Bei einer simulierten Dauer von x und einem Takt der Periode p muss der taktinterne Prozess genau $\lfloor \frac{s}{p} \rfloor$ Mal gestartet werden, damit er den Modulo-n-Zähler erhöhen und die entsprechenden Überprüfungen durchführen kann. Dies ist unabhängig davon, ob ein Prozess eine Taktflanke abgefordert hat. Es ist folglich erstrebenswert, einen Weg zu finden, diese untere Schranke zu durchbrechen.

5.5.1. Lösungsansatz mit IEEE1666-konformem Simulationskernel

Möchte man diese Aktivierungen des Prozesses vermeiden, muss eine andere Möglichkeit gefunden werden, die Erzeugung der abgeforderten Events zum richtigen Zeitpunkt zu garantieren. Dazu folgt ein Grundkonzept.

Die Verwendung einer geordneten Datenstruktur ist unvermeidbar, da bekannt bleiben muss, wann welche Ereignisse relativ zur aktuellen Flanke angefordert sind, und das nächste muss auffindbar sein. Der von Grellier dafür verwendete Ringpuffer ist eine effiziente Möglichkeit, da Abfragen der Ereignisse über simple Index-Operation geschehen können. Suchen nach dem nächsten Ereignis geschehen zwar in $O(n)$, jedoch sind diese n in realen Anwendungen klein (unter 10000). Darüber hinaus wird nach einem Takt nur eine Suche durchgeführt, jedoch werden innerhalb eines Taktes eine Vielzahl von Event-Abfragen (`edge_in_cycle`)

[17] Der Ringpuffer kann bei Anfragen nach späteren Taktflanken vergrößert werden. Die realistische Annahme dabei ist, dass sich sehr schnell nach Simulationsbeginn eine statische Größe des Puffers einstellt.

getätigt, da oft viele Prozesse von einem Takt getrieben werden. Daher ist eine hoch-effiziente Abfrage einer effizienten Suche vorzuziehen.

Geht man also von der Verwendung des Ringpuffers aus, muss eine Möglichkeit gefunden werden, die regelmäßige Aktivierung des Prozesses zu vermeiden, der die Modulo-n-Zählung durchführt und die Ereignisse auslöst. Dies kann erreicht werden, indem der Takt intern den Zeitpunkt seiner letzen Flanke speichert. Dies nenne ich die „lokale Zeit“ des Taktes. Fragt ein Prozess eine Taktflanke mittels `edge_in_cycle(t)` ab, so führt der Takt dabei die lokale Zeit so dicht wie möglich an die aktuelle Simulationszeit heran (siehe Listing 5.12, Zeilen 17 bis 31)[18].

Bei dieser Heranführung wird auch der Modulo-n-Zähler, also der Index im Ringpuffer, um die Anzahl übersprungener Takte erhöht. Dann kann genau wie bei Grelliers Variante das Element $((t+m) \mod n)$ aus dem Ringpuffer zurückgeliefert werden. Neben der lokalen Zeit speichert der Takt intern auch immer den relativ zum aktuellen Wert des Modulo-n-Zählers nächsten, bereits abgefordeten Index `m_relative_next` ab. Dieser wird bei der Nachführung der lokalen Zeit auch entsprechend korrigiert. Nachdem die Zeit nachgeführt wurde, kann der Takt testen, ob `t<m_relative_next` gilt. Wenn dem so ist, wird ein taktinternes Event mittels `notify(t*Periode+lokaleZeit-sc_time_stamp())` ausgelöst. Das heißt genau für den Zeitpunkt, an dem die nächste angeforderte Taktflanke liegen muss. Dies ist in Listing 5.12, Zeilen 32 bis 40 skizziert.

```
1  /*
2  m_pos_timed_events ist der Ringpuffer der Eventzeiger
3    skip() bewegt den Zeiger im Ringpuffer eins nach vorn
4    skip_n(int) bewegt den Zeiger im Ringpuffer n Elemente nach vorn
5    time(t) liefert den Eventzeiger der t+1 Ringpufferelemente weiter vorn liegt
6    set_next_slot(t) setzt den Zeiger im Ringpuffer um t+1 Eintraege nach vorn
7    cycle_immediate() triggert das Event im aktuellen Ringpuffereintrag
8    has_allocated() liefert true wenn mindestens ein zukuenftiges Event  im Ringpuffer vorhanden ist
9    next() liefert den Index des naechsten im Ringpuffer vorhandenen Elements zurueck
10 m_relative_next_tick ist der naechste Takt relativ zum aktuellen der bereits angefordert wurde
11    initial ist dieser auf -1ULL gesetzt
12 m_local_time ist die lokale Zeit des Taktes
13 m_period ist die Periode des Taktes
14 m_stopped steuert, ob der Takt laeuft oder nicht
15 m_long_jump_threshold die Zeitdifferenz ab der eine Division effizienter ist, als eine additive Iteration*/
16
17 void tlm_clock::catch_up(){
18   if (m_stopped!=CLOCK_RUNNING) return;
19   if ((sc_core::sc_time_stamp()-m_local_time)>m_long_jump_threshold){
20     unsigned int distance=(unsigned int)((sc_core::sc_time_stamp()-m_local_time)/m_period);
21     m_local_time+=(m_period* distance);
22     m_pos_timed_events.skip_n(distance);
23     m_relative_next_tick-=distance;
24   }
25   else
26     while(m_local_time+m_period<= sc_core::sc_time_stamp()) {
27       m_local_time+=m_period;
28       m_pos_timed_events.skip();
29       m_relative_next_tick--;
30     }
```

[18]Der Einfachheit der Erklärung halber werden Starts und Stopps von Takten hier nicht erklärt.

```
31 }
32 sc_event& tlm_clock::edge_in_cycle(unsigned t) {
33   catch_up();
34   if (t<m_relative_next_tick && (m_stopped==CLOCK_RUNNING) ) {
35     m_relative_next_tick=t;
36     m_next_edge_time_event.cancel();
37     m_next_edge_time_event.notify(t*m_period+m_local_time-sc_core::sc_time_stamp());
38   }
39   return *m_pos_timed_events.time(t-1);
40 }
41 void tlm_clock::tick(){
42   m_stopped=CLOCK_RUNNING;
43   m_local_time= sc_core::sc_time_stamp();
44   m_pos_timed_events.set_next_slot(m_relative_next_tick-1);
45   m_pos_timed_events.cycle_immediate();
46   m_pos_timed_events.skip();
47   if ( m_pos_timed_events.has_allocated()){
48     m_relative_next_tick=m_pos_timed_events.next();
49     m_next_edge_time_event.notify(m_relative_next_tick*m_period);
50   } else
51         m_relative_next_tick=-1ULL
52 }
```

Listing 5.12: IEEE-1666-konforme Taktimplementierung mit Vermeidung regelmäßiger Prozessreaktivierungen

Dieses Event weckt dann den taktinternen Prozess, welcher den Modulo-n-Zähler auf `m_relative_next` und die lokale Zeit auf die aktuelle Zeit setzt. Danach löst er das Ereignis am aktuellen Index im Ringpuffer aus und sucht anschließend, ob danach ein weiteres bereits angefordertes Ereignis existiert. Ist dem so, berechnet er mit Hilfe der Periode seine nächste Auslösung und setzt `m_relative_next` auf den gefundenen Wert. Ist der Ringpuffer leer, so setzt er `m_relative_next` auf „Unendlich“. Der Prozess ist in Listing 5.12, Zeilen 41 bis 52 dargestellt.

Mit diesem Vorgehen können sehr effizient ungenutzte Takte übersprungen werden. Jedoch ist die Abforderung von Takt-Events teuer. Die Überprüfungen, ob die lokale Zeit nachgeführt werden muss, ob das abgeforderte Ereignis näher liegt als ein bereits abgefordertes Ereignis und die Berechnung der entsprechenden Auslösungszeit des Ereignisses sind aufwändige Operationen. Werden viele oder gar alle Flanke eines Taktes benötigt und werden viele Prozesse von diesem einen Takt getrieben, so dominieren die Kosten der Eventabfrage die durch das effiziente Überspringen erzielbaren Ersparnisse.

Abbildung 5.13 belegt diese Aussagen mit einem einfachen Experiment. Zehn identische Prozesse werden für eine simulierte Zeit von 300 Millisekunden mit einem 100 MHz-Takt betrieben. Die Prozesse überspringen innerhalb von 100 Takten eine zwischen 1 und 100 variierende Anzahl von Takten. In Abbildung 5.13 kann die Abhängigkeit der Simulationslaufzeit von dieser Anzahl übersprungener Takte gesehen werden. Man erkennt die bereits erwähnte untere Schranke der Simulationslaufzeit bei der Taktimplementierung nach Grellier (Messreihe „TI Takt“) und dass diese Schranke bei der alternativen Implementierung nicht existiert. Es kann erkannt werden, dass im Worst-Case (keine Takte werden übersprungen) der alternative Takt eine um ca. 20 Prozent schlechtere Simulationsperformance aufzeigt

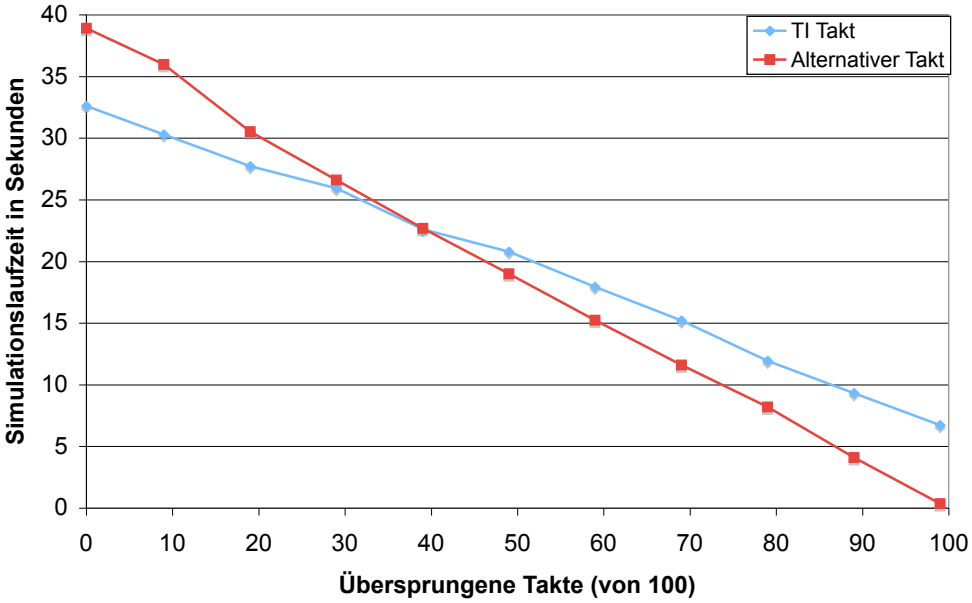


Abbildung 5.13.: Simulationslaufzeiten des Taktes nach Grellier und der vorgeschlagenen Alternative in Abhängigkeit von der Anzahl übersprungener Takte

als die Implementierung nach Grellier. Erst wenn ca. 40 Prozent der Takte übersprungen werden, greifen die positiven Effekte des effizienten Taktüberspringens.

Das bedeutet, dass die Verwendung der alternative Taktimplementierung bei Simulationen mit einer hohen Zahl übersprungener Takt lohnenswert ist, während bei vielen aktiven Takten eher auf die Variante von Grellier zurückgegriffen werden sollte. Im Vorhinein ist bei einer Simulation aber schwer zu beurteilen, wie viele Takte im Mittel übersprungen werden können, speziell wenn man im Rahmen der Performance-Analyse nach Hochlast-Situationen sucht. Darüber hinaus ist die mittlere Zahl übersprungener Takte nicht immer das richtige Maß. Es kann zu Fällen kommen, bei denen über einen langen Zeitraum gar keine Takte übersprungen werden, um danach eine sehr hohe Zahl von Takten zu überspringen. In diesem Fall ist keine der beiden Taktimplementierungen ideal, und ein dynamisches Umschalten der verwendeten Taktimplementierung ist in SystemC nur schwer möglich.

Mit einem IEEE-1666-konformen Kernel konnte keine bessere Lösung gefunden werden. Es bleibt also zu untersuchen, ob und wie der Kernel eine hoch-performante Taktimplementierung unterstützen könnte, die im Wort-Case (jede Flanke wird abgefordert) nicht schlechter ist als die durch Grellier bestimmte Variante, jedoch keine untere Grenze der Simulationsdauer definiert, also eine Simulationdauer von 0s bei 100 Prozent irrelevanter Takte erlaubt.

5.5.2. Modifikation des SystemC-Kernels

Dynamisches Profiling der in Listing 5.12 skizzierten Lösung hat gezeigt, dass bei vielen `edge_in_cycle`-Aufrufen innerhalb eines Zeitschritts die häufigen `sc_time_stamp`-Aufrufe

und auch die arithmetischen Operationen auf den Zeitmarken (in Listing 5.12 Zeilen 19 und 26) hohe Kosten verursachen.

Diese Kosten können signifikant reduziert werden, wenn gespeichert wird, ob zu einem Zeitschritt die lokale Zeit bereits einmal nachgeführt wurde. Bei einem Versuch der Zeitnachführung würde zuerst günstig ein Test stattfinden, und nur wenn dieser negativ ausfällt, wird die kostenintensive Nachführung vollzogen. Danach wird der entsprechende Wert auf `true` gesetzt. Problem dieses Ansatzes ist die Rücksetzung des gespeicherten Wertes. Im nächsten Zeitschritt muss potentiell wieder eine Nachführung stattfinden, der Wert muss also auf `false` gesetzt werden. Innerhalb eines IEEE-1666-konformen SystemC-Simulationskernels ist es aber nicht möglich festzustellen, ob und wann ein Zeitschritt wirklich vorbei ist. Der Zeitpunkt zum Rücksetzen des Wertes ist folglich nicht bestimmbar.

Um dies zu ermöglichen, wird der Simulations-Kernel so verändert, dass er vor der Erhöhung der Simulationszeit jeden Takt informiert, dass der Zeitschritt vorbei ist. Dann kann ein Takt den oben genannten Wert wieder zurücksetzen.

Eine zweite Möglichkeit zur Optimierung des Taktes aus Listing 5.12 liegt in der Planung der Ausführung. In den Zeilen 37 und 49 wird dazu ein Event ausgelöst. Dieses wird dann in die normale Liste der zeitverzögerten Eventauslösungen einsortiert. Die Kosten dieser Operation hängen von der Anzahl der Elemente in dieser Liste ab. Diese Anzahl liegt ausserhalb der Kontrolle der Takte, sodass die Takt-Performance damit stark von äußeren Faktoren abhängig wird. Dies wird erschwert durch die Tatsache, dass abgebrochene Events (`cancel`) wie aktive Events in der Liste verbleiben und erst bei dem Versuch ihrer Auslösung verworfen werden.

Zur Optimierung der Zeitnachführung wurde bereits die Notwendigkeit für einer Benachrichtigung an alle Takte erläutert, wann ein Zeitschritt zu Ende ist. Dieser Aufruf kann auch genutzt werden, um die zweite Situation zu verbessern. In dem Aufruf muss der Simulationskernel nun zusätzlich die aktuell bekannte nächste Zeitmarke, zu der irgendein Takt eine Taktflanke erzeugen will[19] und dazu eine performante Datenstruktur zur Speicherung von Zeigern auf Takte (in Listing 5.14 Zeile 22) übergeben. Jeder Takt kann dann in diesem Aufruf die Zeitmarke der nächsten bereits abgerufenen Taktflanke berechnen[20] und überprüfen, ob diese Zeitmarke weniger weit, genauso weit oder weiter in der Zukunft liegt als die vom Kernel angezeigte Marke. Liegt sie weniger weit in der Zukunft, leert der Takt den übergebenen Container, fügt einen Zeiger auf sich selbst in den Container ein und setzt die vom Kernel übergebene Zeitmarke auf die vom Takt berechnete Marke. Liegt die vom Takt berechnete Marke auf der gleichen Zeit wie die vom Kernel übergebene, so fügt der Takt schlicht einen Zeiger auf sich selbst in den übergebenen Container ein. Liegt die vom Takt berechnete Zeitmarke weiter in der Zukunft als die übergebene Zeitmarke, tut der Takt einfach nichts. Somit hat der Kernel nach Aufruf der Funktion auf allen Takten eine Datenstruktur mit Zeigern auf all die Takte, die am nächsten in der Zukunft eine Taktflanke erzeugen möchten.

[19] Für den ersten informierten Takt innerhalb eines Zeitschrittes wäre dies „Unendlich“.
[20] Mit geschickten Caching dieses Wertes muss dies nicht jedes Mal wirklich berechnet werden.

Dies kann der Kernel dann nutzen, um die nächste zu simulierende Zeitmarke festzulegen und die entsprechenden taktinternen Prozesse zu starten. Dazu ruft er auf allen Zeigern im Container den Callback `tick()` auf. Dieser kann dann die abgerufenen Events auslösen.

Damit ergibt sich für die Planung der nächsten Zeitmarke für Taktflanken bei n-Takten eine Komplexität von $O(n)$ bei jedem Zeitvorschub, da auf jedem Takt eine Operation mit konstanten Kosten ausgeführt wird. Dies ist besser als die $O(n \log n)$-Worst-Case-Komplexität bei der Taktimplementierung mit zeitverzögerten Events (in jedem Zeitschritt sortieren n Takte Events in einen binären Heap mit n Elementen ein), jedoch schlechter als die $O(\log n)$ Best-Case-Komplexität (pro Zeitschritt sortiert nur ein Takt ein Event in einen binären Heap mit n Elementen ein).

Wie später ein Experiment belegen wird (Abbildung 5.20 und deren Erläuterung auf Seite 140), sind die n mal anfallenden realen Kosten bei diesem Ansatz so gering, dass es keine messbaren negativen Auswirkungen gibt.

Zusammenfassend sind die notwendigen Erweiterung des Simulationskernels in Listing 5.14 dargestellt.

```
1  class kernel_to_clock_interface; //Vorwaertsdeklaration
2  //Cache zur effizienten Speicherung von Zeigern auf kernel_to_clock_interface
3  //  alle Zugriffe sollten in O(1) ausfuehrbar sein
4  class clock_ptr_cache{
5    void insert(kernel_to_clock_interface*); //fuege einen Event-Zeiger zum Cache hinzu
6    bool next(); //true, wenn noch mindestens ein Zeiger im Cache ist
7    kernel_to_clock_interface* get(); //zerstoerendes Lesen aus dem Cache
8    void reset(); //Leeren und fuer neue Fuellung vorbereiten
9  };
10
11 //Basisklasse fuer alle Takte
12 class kernel_to_clock_interface{
13   //Diese Funktion registriert einen Takt bei Kernel
14   //  sie muss von einem Takte auf sich selbst aufgerufen werden
15   void add_to_kernel();
16
17   //Diese Funktion meldet einen Takt bei Kernel ab (z.B. wenn er gestoppt wird)
18   //  sie muss von einem Takten auf sich selbst aufgerufen werden
19   void remove_from_kernel();
20
21   //Der Callback, der vom Kernel vor einer Zeiterhoehung auf allen registrierten Takten aufgerufen wird
22   virtual void schedule(clock_ptr_cache&, sc_time&)=0;
23
24   //Der Callback, der vom Kernel auf den Takten aufgerufen wird, die eine Flanke erzeugen sollen
25   virtual void tick()=0;
26 };
```

Listing 5.14: SystemC-Erweiterungen zur Unterstützung von effizienten Taktimplementierungen

5.5.3. Funktionsweise

Da der `clock_ptr_cache` aus Listing 5.14 eine feste obere Grenze von enthaltenen Elementen kennt (nämlich höchstens so viele Elemente wie es Takte im System gibt), ist eine Implementierung, bei der alle Operationen in $O(1)$ ausgeführt werden können, trivial.

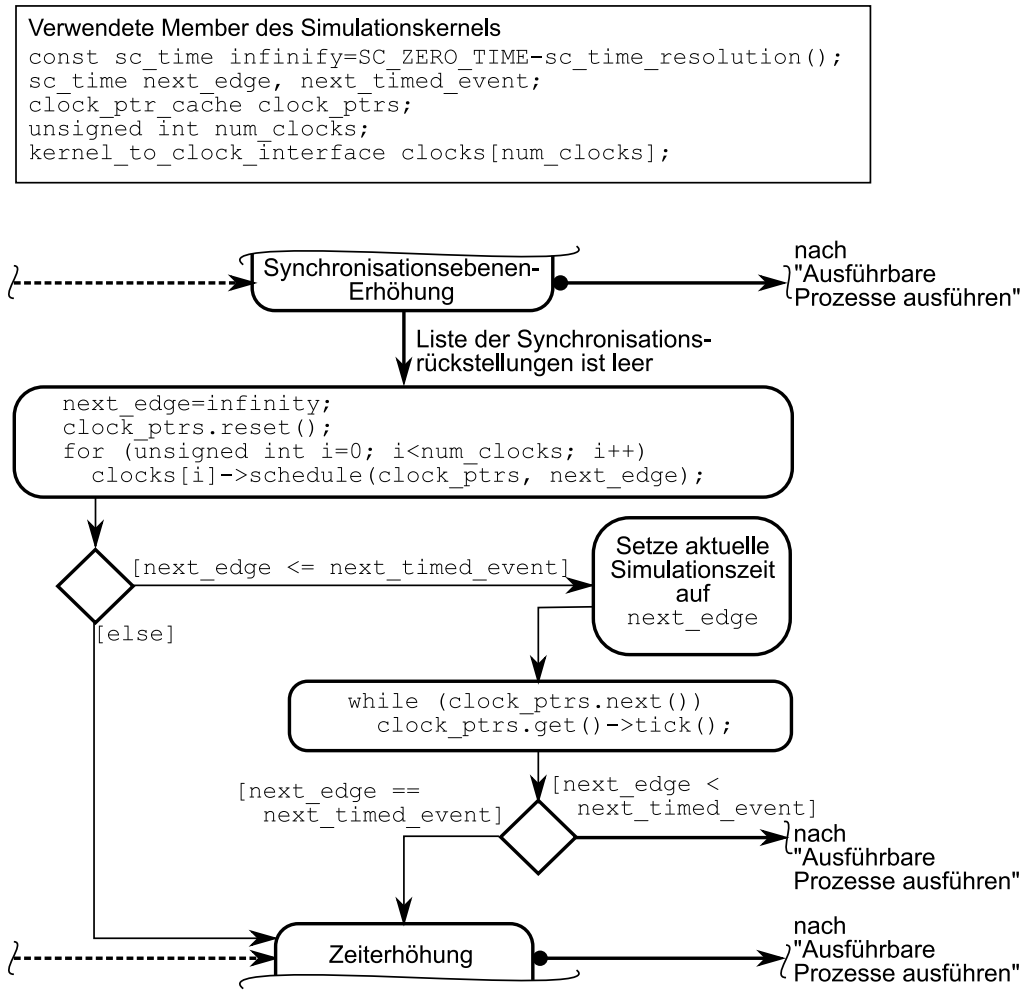


Abbildung 5.15.: Erweiterung der Arbeitsweise des SystemC-Simulationskernels zur effizienten Unterstützung von Taktimplementierungen (Vergleiche dazu Abbildung 5.10)

Die Erweiterung des Simulationskernels ist in Abbildung 5.15 dargestellt und sollte in Verbindung mit Abbildung 5.10 betrachtet werden. Das im vorangegangenen Abschnitt beschriebene Konzept wird direkt umgesetzt: Vor einer Zeiterhöhung iteriert der Kernel über alle vorhandenen Takte (`clocks`), um die Liste aller Takte (`clock_ptrs`) zu erhalten, die am nächsten in der Zukunft (`next_edge`) eine Flanke generieren möchten. Entspricht `next_edge` der Zeitmarke des nächsten regulär zeitverzögerten Events `next_timed_event`, oder liegt `next_edge` davor, wird die aktuelle Simulationszeit auf `next_edge` gesetzt[21], und alle Takte in `clock_ptrs` erhalten ihre `tick`-Callbacks. Gibt es dazu keine regulär zeitverzögerten Events, werden die durch die `tick`-Callbacks ausführbar gewordenen Prozesse ausgeführt. Gab es zum gleichen Zeitpunkt noch regulär zeitverzögerte Events, wird der Kernel diese normal auslösen und dann auch die ausführbaren Prozesse ausführen.

Man erkennt in Abbildung 5.15 sehr gut den bereits erwähnten Overhead einer Iteration über alle Takte bei jedem Zeitvorschub.

[21] Und die globale Synchronisationsebene wird auf Null gesetzt.

5.5.4. Fazit

Die in den Kernel integrierte Unterstützung für Taktimplementierungen erlaubt es, hocheffizient ungenutzte Takt zu überspringen und dabei trotzdem noch eine gute Simulationsperformance zu erzielen, wenn keine oder wenige Takte übersprungen werden. Diese Aussage wird durch die folgenden Experimente belegt.

Zuerst muss gezeigt werden, dass die Modifikation des Kernels die Performance von Simulationen, die keine Takte nutzen, nicht negativ beeinflusst. Dazu wurde ein simples Modul wie in Listing 5.16 für einen Zeitraum von 300 Millisekunden betrieben. 75 Prozent des ausgeführten Codes fallen der Ausführung des Zustandsautomaten des Kernels zu, ein Overhead dabei sollte also messbar sein. Die Simulation wurde mit einem modifizierten und einem unmodifizierten OSCI-Referenzimplementierungs-Kernel durchgeführt. Es konnten keine Laufzeitunterschiede festgestellt werden.

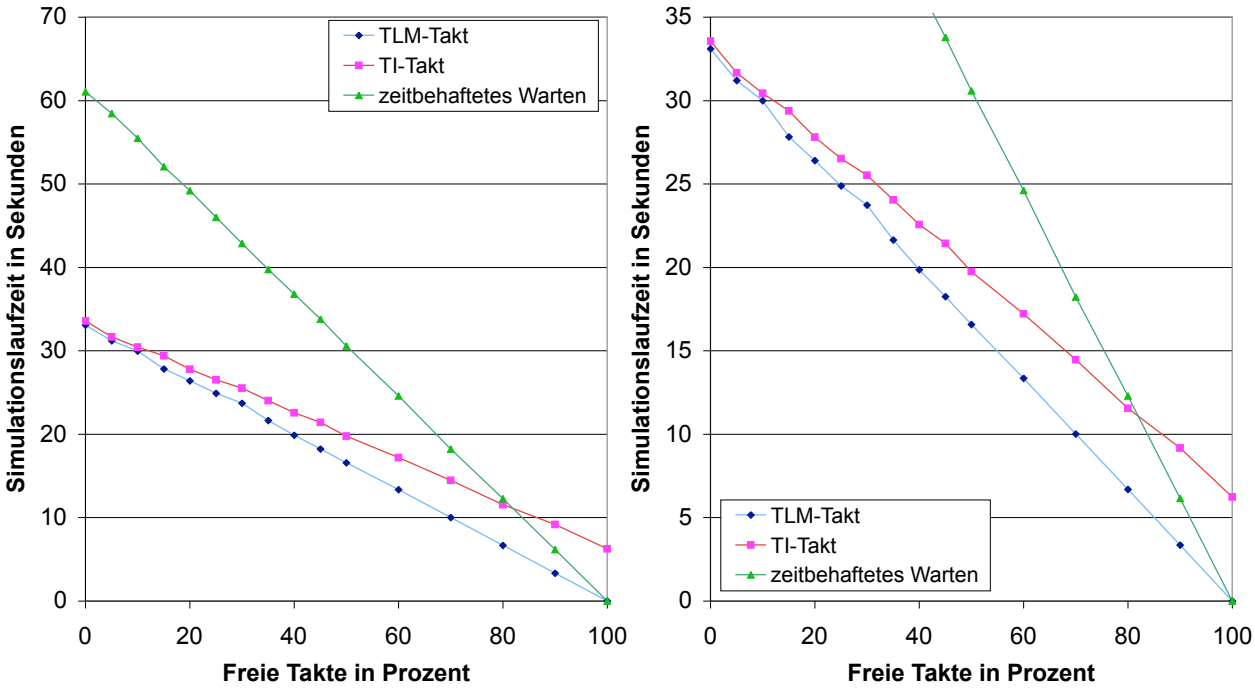
```
1 SC_MODULE(generator){
2   SC_CTOR(generator) : period(10,sc_core::SC_NS){
3     SC_METHOD(run);
4   }
5
6   void run(){next_trigger(period);}
7
8   sc_core::sc_time period;
9 };
```

Listing 5.16: Testmodul zur Evaluation des Overheads der Kernelmodifikation zur Taktunterstützung

Das Experiment mit zehn identische Prozessen, die für eine simulierte Zeit von 300 Millisekunden mit einem 100 MHz-Takt betrieben werden, während die Prozesse innerhalb von 100 Takten eine zwischen 1 und 100 variierende Anzahl von Takten überspringen, wurde wiederholt. Das Ergebnis ist in Abbildung 5.17 auf der nächsten Seite zu sehen. In Abbildung 5.17a werden alle drei möglichen Taktimplementierung vollständig gegenübergestellt, während in Abbildung 5.17b die Skalierung der Y-Achse so gewählt wurde, dass die Verhältnisse zwischen der Taktung nach Grellier (in den folgenden Diagrammen Messreihe „TI-Takt“) und der Taktung mit Kernel-Modifikation (Messreihe „TLM-Takt“) deutlicher werden.

Abbildung 5.17 zeigt, dass die Performance der Taktimplementierung mit Hilfe der Kernel-Modifikation auch im Worst-Case, also bei Anforderung jeder Flanke, mit der von Grellier konkurrieren kann, ungenutzte Takt aber deutlich effizienter überspringt. Der positive Effekt dieses effizienteren Überspringens wird ab ca. 20 Prozent ungenutzten Takten messbar. Der große Vorteil dieses Effekts wird später noch in einem weiteren Experiment und auch im Rahmen der Untersuchungen zum PLB verdeutlicht.

Für ein weiteres Experiment wurden mehrere unabhängige Takte unterschiedlicher Frequenz instanziiert, welche jeweils 50 Module treiben, die immer jede Flanke benötigen. Das Ergebnis zeigt Abbildung 5.18 auf Seite 141. Auch hier skalieren der Takt nach Grellier und der Takt mit Kernel-Modifikation identisch, der Laufzeitunterschied liegt innerhalb der Messungenauigkeit. Man erkennt deutlich die schlechte Skalierung des zeitbehafteten Wartens in



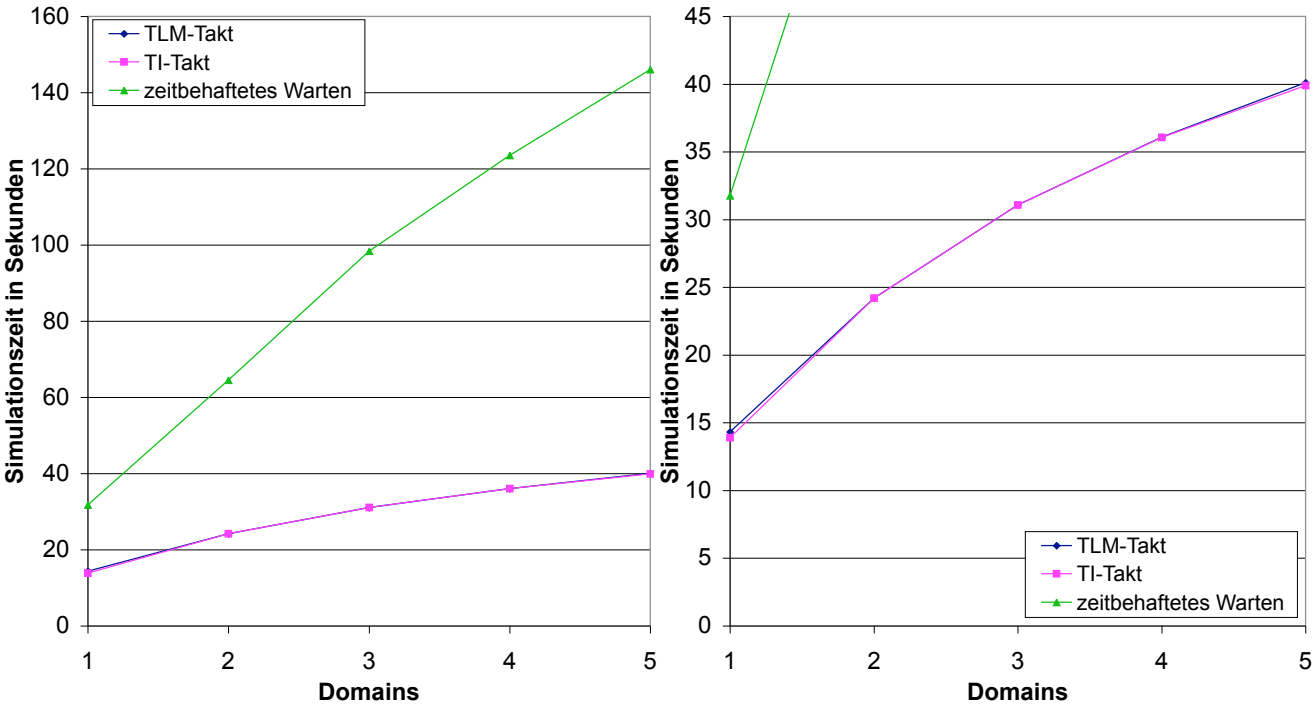
(a) in Relation zu zeitbehaftetem Warten (b) ohne Relation zu zeitbehaftetem Warten

Abbildung 5.17.: Simulationslaufzeiten verschiedener Taktimplementierungen in Abhängigkeit von der Anzahl übersprungener Takte

diesem Fall: Mit jeder Domäne steigt die Anzahl der Events in der Liste der zeitverzögerten Event-Auslösungen um 50, sodass die Einsortierungen mit jeder Domäne sehr viel teurer werden, während bei Grellier diese Anzahl nur um ein Event pro Domäne steigt und bei der Taktimplementierung mit Kernelmodifikation nur ein weiterer Takt in die Liste der Takte aufgenommen wird.

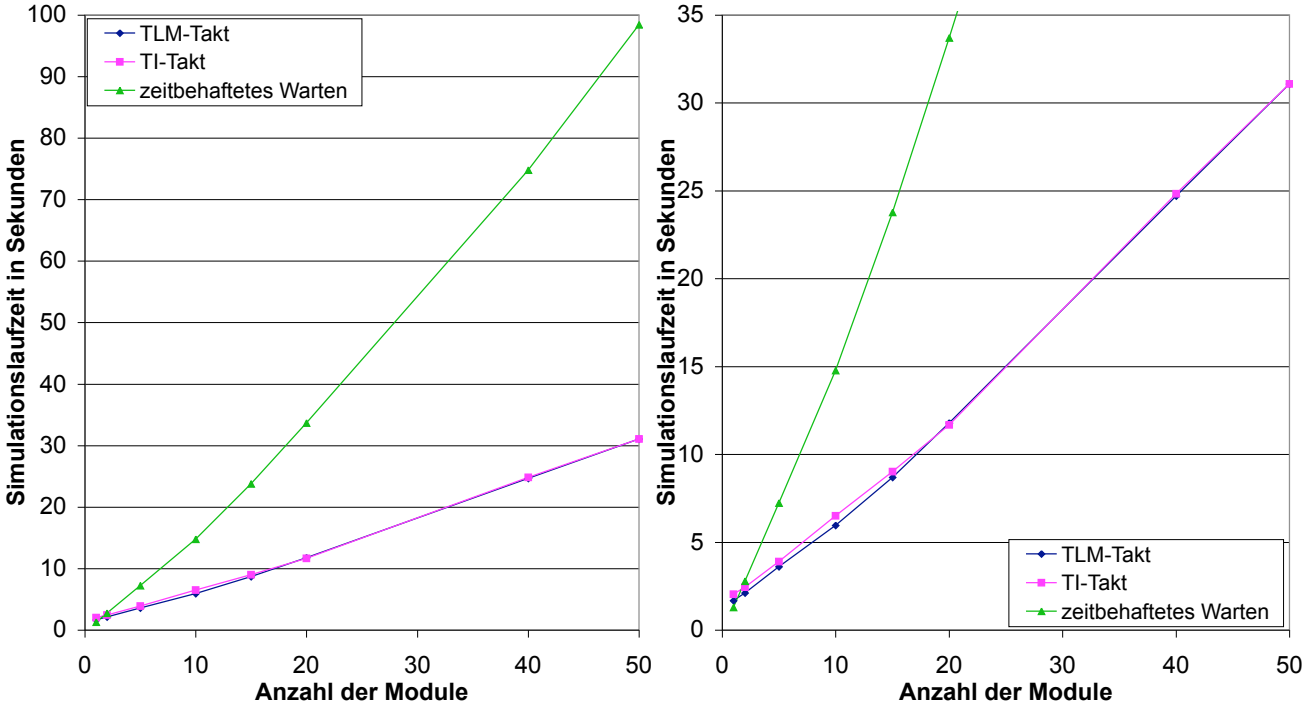
Im gleichen Experiment wurde bei drei Domänen und keinen freien Takten (also wieder dem Worst-Case für die Taktimplementierung mit Kernel-Modifikation) die Anzahl der Module, die vom Takt getrieben werden, erhöht. Es wird also der Overhead des Abrufs eines Events vermessen. Abbildung 5.19 auf der nächsten Seite zeigt, dass die Taktimplementierung mit Kernel-Modifikation und die von Grellier dabei gleichwertig sind. Man erkennt in Abbildung 5.19b auch, dass bei sehr wenigen Modulen (drei Module, eins pro Domäne) die Taktung mit Zeitverzögerungen schneller ist als die anderen beiden Varianten. Jedoch sind nur drei Prozessen in einer realen Simulation recht unwahrscheinlich.

Danach wurde der Test wieder auf eine Takt-Domäne reduziert, in dieser wurde dann aber der Takt mehrfach halbiert. Jeder dieser geteilten Takte treibt zehn Prozesse. Die Teilung geschah im Falle der Taktimplementierung mit Kernel-Modifikation mit Hilfe eines entsprechenden Takt-Synchronisierers (siehe auch Abschnitt 4.10.3) und bei der Taktimplementierung nach Grellier mit den von ihm implementierten Takt-Teilern. Für die Taktmodellierung mit zeitbehaftetem Warten wurde die zu wartende Periode schlicht verdoppelt. Abbildung 5.20 auf Seite 142 zeigt die Simulationsperformance in Abhängigkeit von der Anzahl der kaskadierten Taktteilungen. Man erkennt, dass die Taktimplementierung nach



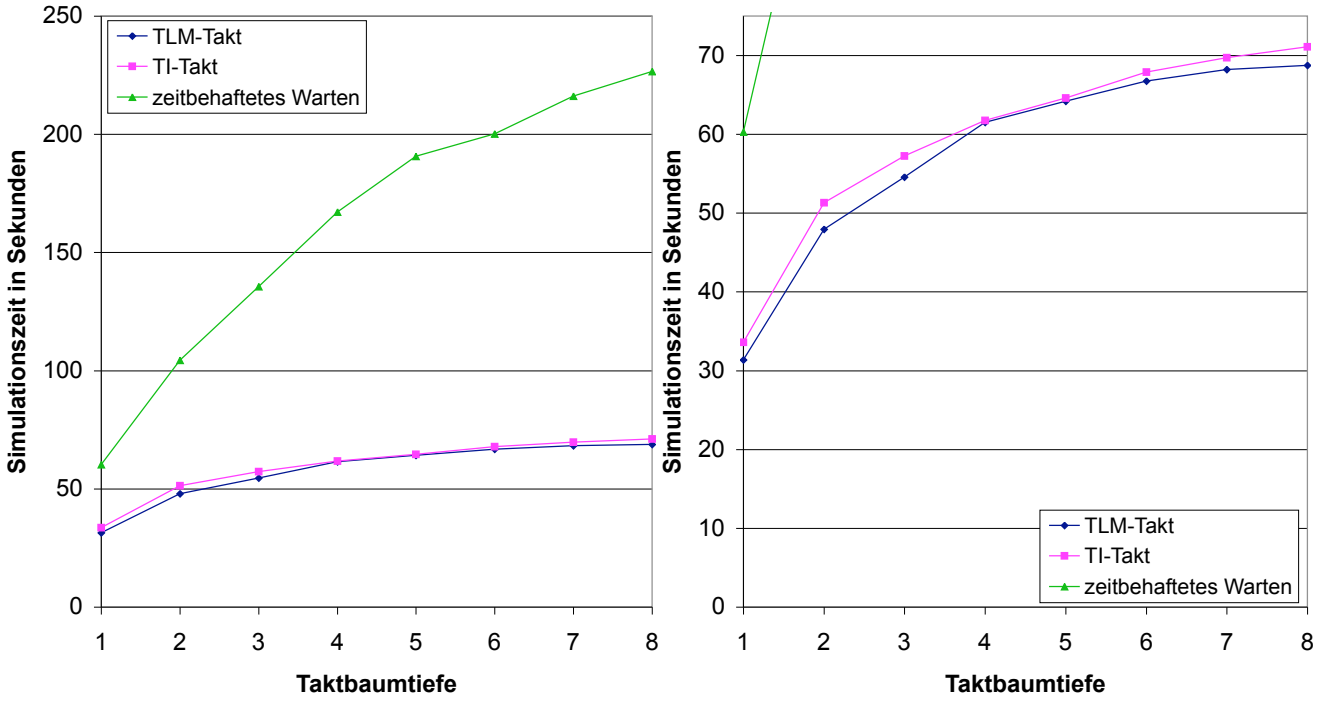
(a) in Relation zu zeitbehaftetem Warten (b) ohne Relation zu zeitbehaftetem Warten

Abbildung 5.18.: Simulationslaufzeiten verschiedener Taktimplementierungen in Abhängigkeit von der Anzahl paralleler Takt-Domänen



(a) in Relation zu zeitbehaftetem Warten (b) ohne Relation zu zeitbehaftetem Warten

Abbildung 5.19.: Simulationslaufzeiten verschiedener Taktimplementierungen in Abhängigkeit von der Anzahl vom Takt getriebener Prozesse



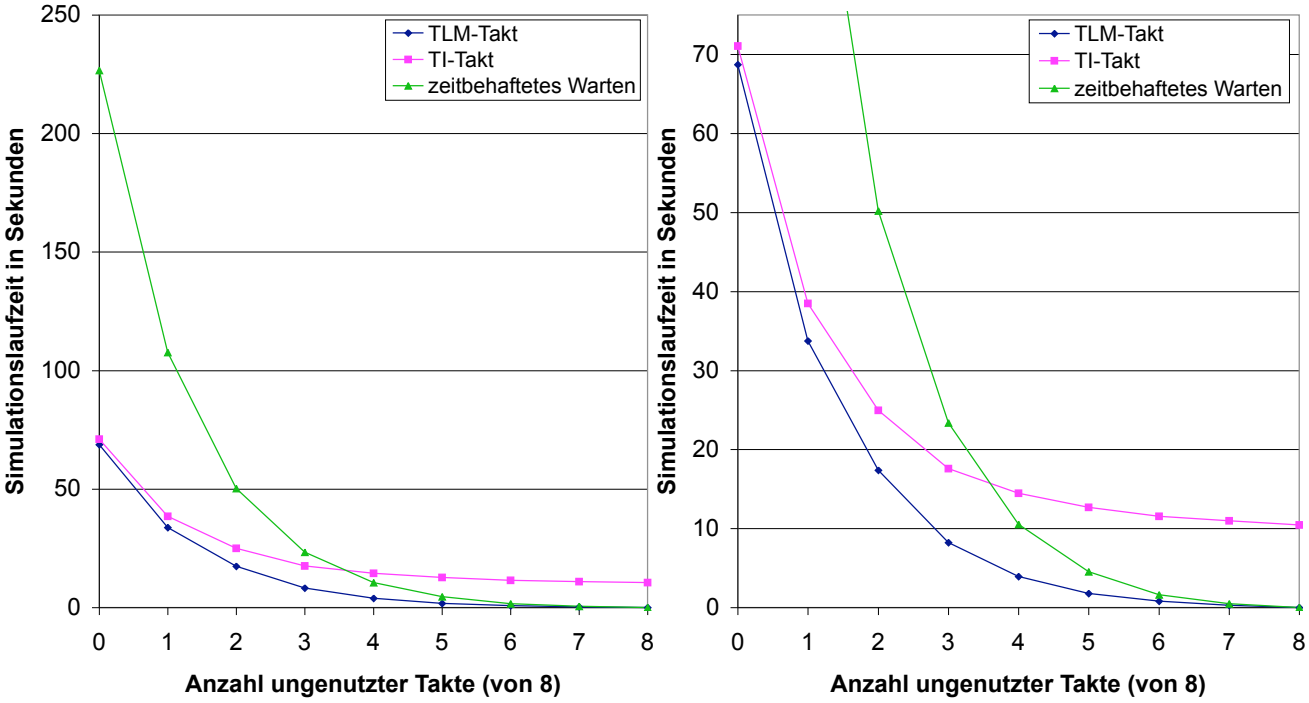
(a) in Relation zu zeitbehaftetem Warten (b) ohne Relation zu zeitbehaftetem Warten

Abbildung 5.20.: Simulationslaufzeiten verschiedener Taktimplementierungen in Abhängigkeit von der Anzahl kaskadierter Taktteiler

Grellier und mit Kernel-Modifikation wieder gleichwertig sind. Dieser Test ist dahingehend sehr interessant, dass zum Beispiel die Instanziierung des achten Taktes als Ergebnis der siebten Taktteilung dazu führt, dass bei der Taktimplementierung mit Kernel-Modifikation nun bei jedem Zeitvorschub über acht anstatt sieben Takte iteriert wird. Da es 30 Millionen Zeitvorschübe gibt (der erste Takt im Taktbaum hat eine Periode von 10 ns und die simulierte Dauer sind 300 ms) und nur jeder 128te Zeitvorschub mit einer Flanke des achten Taktes zusammenfällt, bedeutet dies ca. 29,77 Millionen prinzipiell unnötige Aufrufe von `schedule` auf diesem achten Takt. Bei Grellier wird ein geteilter Takt immer von jeder zweiten Flanke des Vorgängers getrieben; nur der erste Takt muss wirklich zeitbehaftete Auslösungen vornehmen. Somit werden die taktinternen Prozesse über normale Events gestartet. Dies geschieht für den achten Takt 234.375 Mal. Offenbar sind die Kosten dieser 234.375 Event-Auslösungen und Prozessaktivierungen teurer als die 29,77 Millionen unnötigen Aufrufe von `schedule`. Dies stützt meine Annahme, dass die $O(n)$-Komplexität bei jedem Zeitvorschub keinen realen Nachteil darstellt.

In einer Variation des Tests mit Taktteilern gibt es wiederum einen Takt, der mehrfach geteilt wird, jedoch treibt diesmal nicht jeder (Zwischen-)Takt Module. Eine variable Anzahl von Takten treibt nur den Folgetakt. Im ersten Durchlauf treiben alle Takte Prozesse, im zweiten Durchlauf treibt der erste Takt der Teilerkaskade nur noch den Folgetakt, im dritten Durchlauf treiben die ersten beiden Takte der Kaskade nur noch ihren Nachfolger und so weiter. Im letzten Durchlauf treibt nur noch der letzte Takt in der Teilerkaskade Prozesse. Man erkennt in Abbildung 5.21 auf der nächsten Seite sehr gut, dass die Taktimplementie-

rung mit Kernel-Modifikation nun ihre Vorteile nutzt. Ein ungenutzter Takt verursacht kaum messbaren Overhead, während bei Grellier ein geteilter Takt die Erzeugung jeder zweiten Flanke des Vorgängertaktes erzwingt. Der Performance-Vorteil der Taktimplementierung mit Kernel-Modifikation beträgt bei nur zwei ungenutzten Takten bereits 25 Prozent.



(a) in Relation zu zeitbehaftetem Warten (b) ohne Relation zu zeitbehaftetem Warten

Abbildung 5.21.: Simulationslaufzeiten verschiedener Taktimplementierungen in Abhängigkeit von der Anzahl ungenutzter aktiver Takte

Schließlich sollte noch der Overhead von deaktivierten, also gestoppten Takten vermessen werden. Bei der Taktimplementierung nach Grellier verursacht ein gestoppter Takt keinerlei Overhead, da der taktinterne Prozess auf ein niemals ausgelöstes Event wartet. Bei der Taktimplementierung mit Kernel-Modfikation entfernt sich ein gestoppter Takt aus der Liste der im Kernel registrierten Takte und wird so bei einem Zeitvorschub nicht mehr abgefragt. Um dies zu Belegen, wurde ein aktiver Takt mit zehn getriebenen Prozessen sowie eine variable Anzahl von gestoppten Takten erzeugt und die Laufzeit für 30 simulierte Millisekunden vermessen. Abbildung 5.22 auf der nächsten Seite zeigt das Ergebnis der Messung[22]. Die Laufzeiten schwanken innerhalb der Messungenauigkeit. Wie erwartet gibt es also keinen messbaren Einfluss von gestoppten Takten auf die Simulations-Performance.

Alle durchgeführten Tests zeigen, dass die Taktimplementierung mit Kernel-Modifikation im Worst-Case, wenn jede Flanke abgefordert wird, eine mit der Taktimplementierung nach Grellier vergleichbare Performance zeigt. Das Überspringen von Takten ist effizienter und die bei Grellier existierende untere Schranke für Simulationslaufzeiten existiert nicht. Die

[22]Da es für die Taktung mit Hilfe von zeitbehaftetem Warten keine Entsprechung eines gestoppten Taktes gibt, wurde dies nicht vermessen.

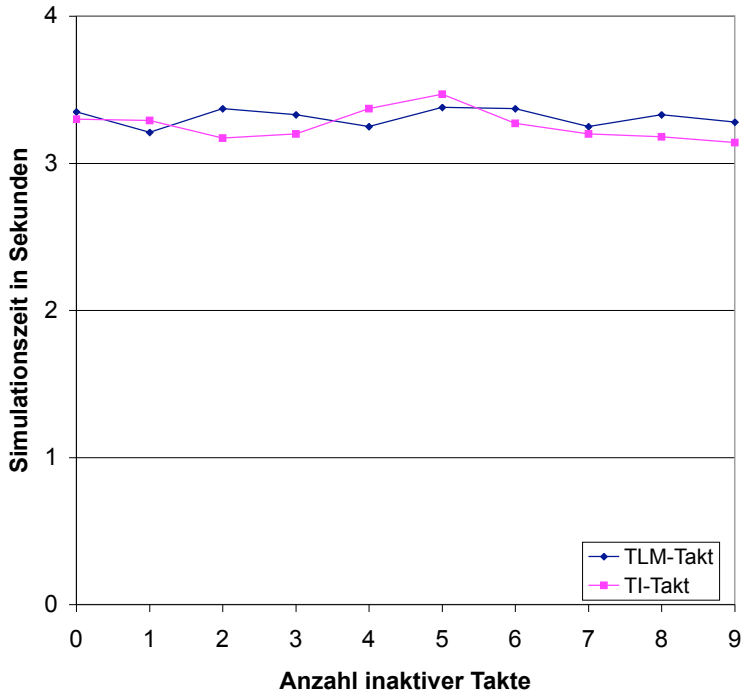


Abbildung 5.22.: Einfluss gestoppter Takte auf die Simulationslaufzeit

Taktimplementierung mit Kernel-Modifikation kann also in jedem Anwendungsfall den Alternativen vorgezogen werden. Nichtsdestotrotz mag sich die Frage stellen, warum so viel Arbeit und Experimente in die Taktimplementierung investiert wurden. Dies wird einerseits bei den Experimenten zum PLB noch deutlich, soll hier aber auch an einem anderen realistischen Beispiel illustriert werden.

Wie bereits in Abschnitt 3.2 erläutert, werden taktgenaue TLM-Modelle häufig in Mixed-Mode-Simulationen verwendet. Zum Beispiel können an einem abstrakten ISS einige taktgenaue Peripherie-Komponenten angeschlossen sein, um die Best-Case- oder Worst-Case-Antwortzeit dieser Peripherie auf verschiedene von der Software durchgeführte Zugriffe zu vermessen, um so z.B. gute Schätzungen für die notwendige Prozessortaktung zu tätigen. In Abbildung 5.23 auf der nächsten Seite ist ein solcher Fall etwas vereinfacht skizziert. Ein OpenRISC 1000 Prozessor [Lamp06]$_i$, modelliert mit Hilfe eines ISS [Benn09]$_i$, wird mit einem Modell eines UART, einer RS232-Verbindung und eines Terminals gekoppelt[23], um zu untersuchen, welchen Einfluss die Baudrate und der Kommunikations-Modus[24], deren dynamische Beeinflussung und gegebenenfalls sogar Kommunikationsstörungen zu verschiedenen Programmausführungszeitpunkten auf die Gesamtlaufzeit und Stabilität der auf dem ISS laufenden Software hat. Dabei soll das Modell des UART mit einem Takt versorgt werden, aus dem es gemäß seines Divisior-Registers die entsprechend gewünschte Baudrate erzeugen kann, mit der es Symbole auf das Modell des RS232-Channels gibt. Der Takt entspricht dabei dem Takt des OpenRISC-Prozessors.

An dieser Stelle stehen nun weniger die Ergebnisse der Untersuchungen, sondern vielmehr die dafür notwendige Simulationsdauer im Vordergrund. Diese hängt stark von der

[23] Der Aufbau basiert auf [Benn10]$_i$ wurde aber erweitert [Günz10a].
[24] IRQ- oder Polling-basiert.

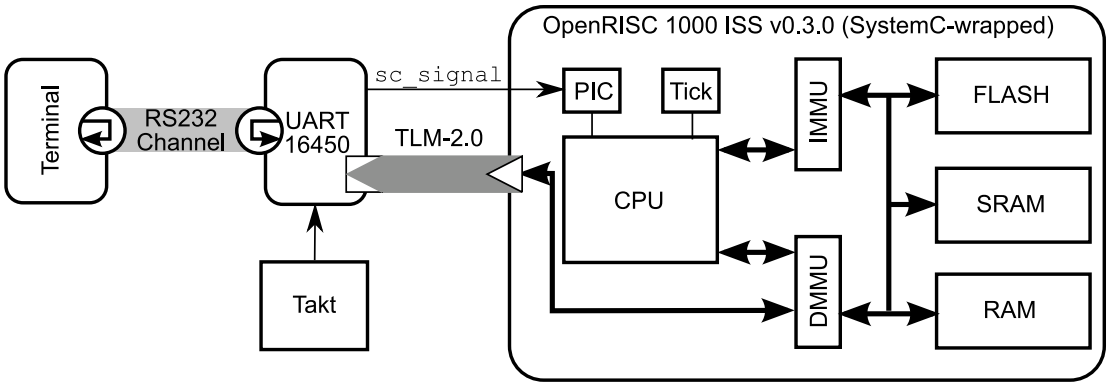


Abbildung 5.23.: Aufbau des Co-Simulations-Experiments

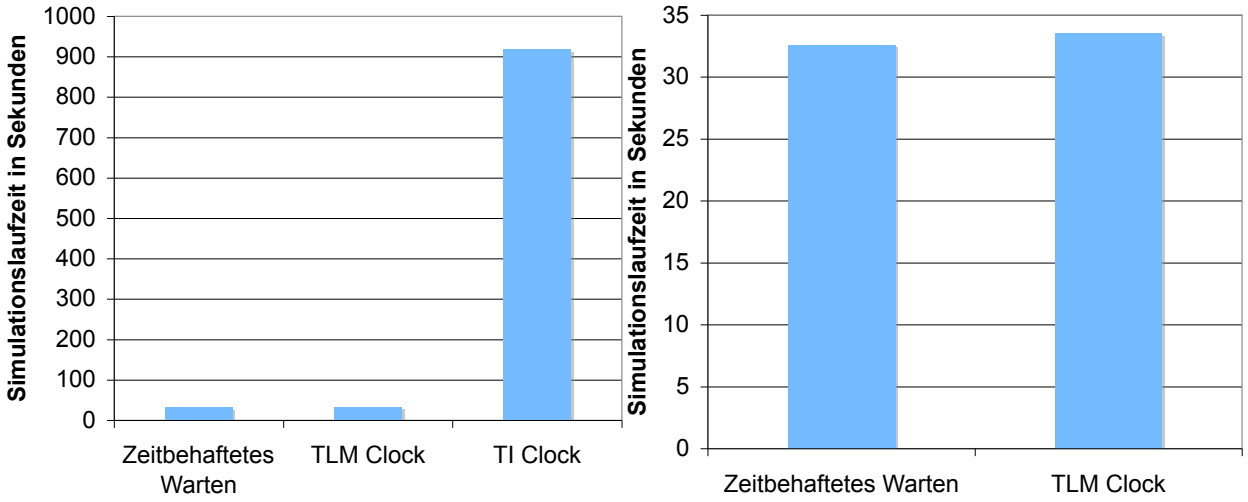
verwendeten Taktimplementierung ab. Dazu habe ich die drei bereits oben gezeigten Implementierungen (nach Grellier, mit Kernel-Modifikation und mit Hilfe von zeitbehaftetem Warten) untersucht. Bei der Implementierung nach Grellier oder mit der Kernel-Modifikation werden die Verzögerungen, also der Zähler im UART mit `edge_in_cylce` und im Falle des zeitbehafteten Wartens mit `next_trigger(sc_time)` erzeugt.

Bei einem 100 MHz-Takt und einer Ziel-Baudrate von zum Beispiel 115,2 kBaud wird von der Software das Divisor-Register auf 54 gesetzt[25]. Dann wird für die Verzögerung eines Characters (10 Bits) `edge_in_cycle(8640)` bzw. `next_trigger(86400 ns)` verwendet. Als Benchmark-Anwendung wurde das Booten eines Linux-v2.6.23-Kernels verwendet, gemessen vom Simulationsstart bis zur Anzeige des Nutzer-Prompts auf dem simulierten Terminal.

Abbildung 5.24 auf der nächsten Seite zeigt die gemessenen Simulationszeiten. Es wird deutlich, dass das effiziente Überspringen ungenutzter Takte in diesem Fall von großem Wert ist. Der UART wird vom Prozessor nur sporadisch genutzt, die Anzahl der nicht genutzten Takte ist hoch, da der UART das einzige Modul ist, das Takte benötigt. Es ergibt sich zum zeitbehafteten Warten fast kein Unterschied in der Performance der Taktung mit Kernel-Modifikation. Es wird hier deutlich, dass die Taktung mit Kernel-Modifikation also auch im Extremfall von sehr wenig genutzten Takten und nur sehr wenig getriebenen Prozessen optimale Performance zeigt. Zusammenfassend kann man sagen:

- Die Taktung nach Grellier ist gut geeignet, wenn sehr wenige Takte übersprungen werden (Abbildung 5.17).
- Die Taktung nach Grellier ist ungeeignet, wenn mäßig bis viele Takte übersprungen werden (Abbildungen 5.17 und 5.24).
- Die Taktung mit Zeitverzögerungen ist optimal geeignet, wenn sehr wenige Module getrieben werden (Abbildungen 5.19 und 5.24).
- Die Taktung mit Zeitverzögerungen ist ungeeignet, wenn viele Module getrieben werden (Abbildung 5.19).

[25]Bei einem UART 16550 wird die Frequenz durch das 16-fache des Divisor-Regsiters geteilt: 54*16=864 Takte → 864*10 ns=8640ns→115,74kBaud



(a) in Relation zur Taktung nach Grellier (b) ohne Relation zur Taktung nach Grellier

Abbildung 5.24.: Simulationslaufzeiten des Linux-Boot-Vorganges auf dem LT-ISS bei Co-Simulation mit einem getakteten Modul in Abhängigkeit von der Taktung

- Die Taktung nach Grellier ist gut geeignet, wenn parallele Takt-Domänen existieren (Abbildung 5.18).
- Die Taktung mit Zeitverzögerungen ist ungeeignet, wenn parallele Takt-Domänen existieren (Abbildung 5.18).
- Die Taktung nach Grellier ist gut geeignet, wenn kaskadierte Takt-Arithmetik (z.B. Taktteilung) vorkommt (Abbildung 5.20).
- Die Taktung mit Zeitverzögerungen ist ungeeignet, wenn kaskadierte Takt-Arithmetik (z.B. Taktteilung) vorkommt (Abbildung 5.20).
- Die Taktung nach Grellier ist ungeeignet, wenn aktive Takte keinen Abnehmer haben (Abbildung 5.21).
- Die Performance der Taktung mit Kernel-Modifikation ist in allen Fällen mit dem jeweiligen Optimum vergleichbar oder stellt selbst das Optimum dar.

Diese Zusammenfassung macht deutlich, dass die Taktimplementierung mit Kernel-Modifikation über alle Anwendungsfälle hinweg die beste Wahl darstellt. Dies ist extrem wichtig, da bei a priori unbekannten Lastsituationen[26] ein Takt verwendet werden muss, der – egal was passiert – stets optimale Simulationsperformance bietet. Die Existenz eines getakteten Moduls im System ist nicht mehr zwingend ein Simulationsperformance-Killer. Handelt es sich um ein reaktives Modul, wie den UART aus obigem Beispiel, ist dessen Existenz prinzipiell straffrei.

[26]Im Rahmen der Performance-Evaluation oder Power-Analyse soll die Simulation die Frage beantworten, ob die entsprechenden Stimuli zu hoher oder geringer Last führen. Die Frage ob es nun überhaupt zu vielen ungenutzten Takten kommt ist somit im Vorhinein nicht klar zu beantworten.

6. Praktische Anwendung

Inhalt

6.1. Einleitung

Dieses Kapitel setzt sich mit der Erfüllung von Zielstellung Z4 aus Abschnitt 3.4 auseinander: Erläuterung des Modellierungsstils an einem respräsentativen Beispiel und Aufzeigen des erzielbaren Speed-Ups. Die Definition von Busphasen wurde von mir zur Entwicklung und Untersuchung meiner Ansätze für alle Busse der AMBA Familie (APB, AXI, AHB und AHBLite), für das OCP und für den PLB und den OPB aus der CoreConnect Familie durchgeführt. Die Ergebnisse der daraus für AMBA und OCP resultierenden GP- und TLM-Phase-Mappings, der Festlegung der Modifiabilities, der Bindungschecks und der protokollspezifischen Regeln finden sich in $[\text{OCP-09}]_i$ bzw. $[\text{CaGr10}]_i$[1]. Die Bestimmung der Busphasen für den OPB ist Anhang D zu entnehmen. Im Folgenden werden die Busphasen des PLB und deren Mappings präsentiert. Mit dem dabei entstehenden TLM-2.0-Interface wird ein taktgenaues busphasenbasiertes *J-R*-Simulaionsmodell des PLB erstellt, und der Speed-Up gegenüber einer RTL-Implementierung wird untersucht. Für alle Experimente gelten grundsätzlich die Rahmenbedingungen gemäß Abschnitt 5.2. Im Fall der RTL-Simulationen wurde Linux Run-Level 1 mit manuell aktivierter Netzwerkunterstützung verwendet, da die Abfrage eines Lizenzservers notwendig war. Aus Fairnessgründen wurden die entsprechenden TLM-Messungen ebenso durchgeführt.

6.2. GP- und TLM-Phase-Mapping des PLB v4.6

Für die Bestimmung von Busphasen war es nicht sinnvoll, allgemeingültige Regeln zu definieren. Die Wahl der Busphasen hängt zu stark vom gewünschten Anwendungsfall (z.B. Performance-Analyse) und den Eigenschaften des zu modellierenden MMBIFs ab. Aus diesem Grund wird in diesem Abschnitt anhand des CoreConnect PLB und im Anhang I anhand

[1] Die in $[\text{OCP-09}]_i$ bzw. $[\text{CaGr10}]_i$ verwendeten Ansätze weichen ggf. in Details von den in dieser Arbeit präsentierten ab, da es sich einerseits um Arbeiten handelt, die erst zu verschiedenen Erkenntnissen geführt haben und da andererseits in industriellen Kooperationen noch weitere Randbedingungen zu berücksichtigen sind, wie z.B. ein unmodifiziertes TLM-2.0 zu verwenden.

des AMBA AHB illustriert und begründet, wie Busphasendefinitionen für die Performance-Analyse, dem meines Erachtens nach wichtigsten Anwendungsfall, festgelegt werden können. Wie der Anwendungsfall der Power-Analyse die Festlegung der Busphasen beeinflusst, wird in Anhang D skizziert. Anschliessend werden dann mit Hilfe des GP- und TLM-Phase-Mappings aus den Busphasendefinitionen TLM-Interfaces für die taktgenaue, busphasenbasierte *J-R*-Simulation bestimmt.

In diesem Abschnitt wird das GP- und TLM-Phase-Mapping des CoreConnect PLB in der Version 4.6 präsentiert. Anhang I tut das gleiche, jedoch bei weitem detaillierter für den AHB. Das PLB-Protokoll ist in seinen Phasen etwas einfacher gestaltet als das AHB-Protokoll. Aus diesem Grund werden die Phasen nur textuell erläutert. Ports werden ähnlich wie in Anhang I der Übersichtlichkeit halber nur mit ihrem Namen anstelle des Tupels genannt. Richtungen und Bitbreiten der genannten Ports können [IBM-04] entnommen werden. So steht zum Beispiel Mn_Request stellvertretend für das Tupel (Mn_Request, 1, e)[2].

Wie bereits erwähnt, ist die Zielstellung hier, ein Interface zur Performance-Evaluation zu bestimmen. Die Grundannahme dabei ist, dass nie ein Modul gegen das PLB-Protokoll verstößt. Somit sollten die zu definierenden Busphasen den Kommunikationsphasen entsprechen und dabei alle Signale, deren Verlauf durch das Protokoll fest vorgegeben ist, implizit modellieren. Dadurch wird von Kommunikationsfehlern innerhalb von Kommunikationsphasen abstrahiert, da während einer Busphase nicht protokollkonforme Änderungen der Busphasenports unmöglich werden.

Tabelle 6.1 zeigt eine Übersicht aller von mir für die Performance-Analyse identifizierten Busphasen des PLB v4.6 mit 32-Bit-Adresse und 64-Bit-Datenbreite[3]. Zur Erinnerung: Eine Busphase Φ wird bestimmt durch die Phasenstartports $PP_{\Phi S}$, die Phasenendports $PP_{\Phi E}$, die Phasenobservierungsports $PP_{\Phi O}$, die Beobachtungsdauer tb_Φ und die Start-, End- und Abbruchkriterien S_Φ, E_Φ, A_Φ. In der Tabelle werden $PP_{\Phi S}$ und $PP_{\Phi E}$ für die jeweiligen Phasen genannt. Die tb_Φ und die eventuell nicht-leere Menge $PP_{\Phi O}$ werden in der weiter unten folgenden Diskussion von S_Φ, E_Φ und A_Φ bestimmt. Darüber hinaus wird dort auch die eigentliche Bedeutung der Phasen erläutert.

Der Defaultwert aller Ports außer Mn_MSize bzw. Sl_SSize, Mn_[r|w]Burst bzw. PLB_[r|w]Burst und Mn_BE ist Null und wird nicht extra gelistet. Für Mn_MSize bzw. Sl_SSize entspricht der Default der Codierung der Breite des Datenbusses am jeweiligen Master oder Slave, für Mn_BE ist der Default Eins für alle Bits des Ports. Für Mn_[r|w]Burst bzw. PLB_[r|w]Burst ist der Default Eins, da nur Änderungen auf Null wichtig sind (siehe unten). Index (SL) in der ersten Spalte von Tabelle 6.1 bedeutet, dass die Phase ohne bzw. mit diesen Zusatz für Master bzw. Slaves existiert. Die Phasenstartports bzw. -endports ergeben sich in diesem Fall für die Phase im Master-Interface (zum Beispiel REQ) aus den in der Spalte „Master“ gelisteten Ports und für das Slave-Interface (zum Beispiel REQ_{SL}) aus den in der Spalte „Slave“ gelisteten Ports. Ansonsten gibt es die Phase nur in dem Interface

[2]Einige Portnamen sind aus Gründen der Übersichtlichkeit leicht abgekürzt.

[3]Um das Beispiel nicht unnötig zu verkomplizieren, wurde auf die optionale Parity-Signalisierung verzichtet.

(Master oder Slave), in dessen Spalte Ports gelistet sind. Ist der Name eines Ports eckig eingeklammert, so ist es ein optionaler Port im Interface. Um später den verhandelbaren Satz von GP-Erweiterungen und TLM-Phasen bestimmen zu können, wird das GP- und TLM-Phase-Mapping später mehrfach, einmal für jede denkbare Konfiguration, durchgeführt. Die Nennung des GP-Mappings in Tabelle 6.1 ist an dieser Stelle noch ein Vorgriff und wird später erklärt.

Phase	Port-gruppe	Master	Slave	GP-Mapping
REQ$_{(SL)}$ (Multi-takt)	PP_S	[Mn_abort]	PLB_abort	implizit
		Mn_request		implizit
			PA_Valid	GP-Erw. `is_prim`, Typ `bool`
			SA_Valid	GP-Erw. `is_sec`, Typ `bool`
		[Mn_priority]	PLB_priority	GP-Erw. `priority`, Typ `short`
		[Mn_busLock]	PLB_busLock	GP-Erw. `is_locked`, Typ `bool`
		Mn_RNW	PLB_RNW	GP-Grundelement `m_command`
		Mn_BE	PLB_BE	GP-Grundelement `m_byte_enable[_length]`
		[Mn_size]	PLB_size	GP-Erw. `trans_size`, Typ `short`
		[Mn_type]	PLB_type	GP-Erw. `trans_type`, Typ `short`
		[Mn_MSize]	PLB_MSize	GP-Erw. `mst_size`, Typ `short`
		[Mn_TAttr]	PLB_TAttr	GP-Erw. `attribs`, Typ `short`
		[Mn_lockErr]	PLB_lockErr	GP-Erw. `lck_err_reg`, Typ `bool`
		Mn_ABus	PLB_ABus	GP-Grundelement `m_address`
			PLB_mstID	GP-Erw. `master_id`, Typ `short`
	PP_E und PP_S	PLB_tOut		GP-Erw. `end_type`, Typ `enum`[4]
		PLB_AAck	Sl_AAck	GP-Erw. `end_type`, Typ `enum`[4]
		PLB_rearb	[Sl_rearb]	GP-Erw. `end_type`, Typ `enum`[4]
	PP_E	PLB_SSize	[Sl_SSize]	GP-Erw. `slv_size`, Typ `short`
RESP$_{(SL)}$ (Einzel-takt)	PP_S	PLB_rDBus	Sl_rDBus	GP-Grundel. `m_data[_length]`
		PLB_rWdA	[Sl_rWdA]	GP-Erw. `rdAddr`, Typ `short`
		PLB_rDAck	Sl_rDAck	implizit
		PLB_MrErr	[Sl_MrErr]	GP-Grundel. `m_response_status`
RBT$_{(SL)}$ (Einzel-takt)	PP_S	PLB_rBTerm	[Sl_rBTerm]	implizit
RBC$_{(SL)}$ (Einzel-takt)	PP_S	[Mn_rBurst]	PLB_rBurst	implizit

[4] `enum end_type_enum{ack, timeout, rearbitrate};`

$DATA_{(SL)}$ (Multitakt)	PP_S	Mn_wDBus	PLB_wDBus	GP-Grundel. `m_data[_length]`
	PP_E und PP_S	PLB_wDAck	Sl_wDAck	implizit
	PP_E	PLB_MwErr	[Sl_MwErr]	GP-Grundel. `m_response_status`
$DBT_{(SL)}$ (Einzeltakt)	PP_S	PLB_wBTerm	[Sl_wBTerm]	implizit
$DBC_{(SL)}$ (Einzeltakt)	PP_S	[Mn_wBurst]	PLB_wBurst	implizit
WAIT (Einzeltakt)	PP_S		[Sl_wait]	implizit
RCOMP (Einzeltakt)	PP_S		Sl_rComp	implizit
DCOMP (Einzeltakt)	PP_S		Sl_wComp	implizit
RPRIM (Einzeltakt)	PP_S		PLB_rPrim	implizit
DPRIM (Einzeltakt)	PP_S		PLB_wPrim	implizit
IRQ_{SL} (Multitakt)	PP_S	PLB_MIRQ	[Sl_MIRQ]	GP-Erw. `irq_set`, Typ `int`
BSY_{SL} (Multitakt)	PP_S	PLB_MBusy	Sl_MBusy	GP-Erw. `bsy_set`, Typ `int`
$DPND_{SL}$ (Multitakt)	PP_S	PLB_wPdRq	PLB_wPdRq	implizit
		PLB_wPdPri	PLB_wPdPri	GP-Erw. `d_pd_pri`, Typ `int`
$RPND_{SL}$ (Multitakt)	PP_S	PLB_rPdRq	PLB_rPdRq	implizit
		PLB_rPdPri	PLB_rPdPri	GP-Erw. `r_pd_pri`, Typ `int`

Tabelle 6.1.: Busphasen des PLB und deren Start- und Endports

In Anhang I werden alle Start-, End- und Abbruchkriterien ausführlich formal beschrieben und daran der Formalismus der Erfassung der Busphasen verdeutlicht. Dabei wird deutlich, dass die formale Notation schwer handhabbar ist. Aus diesem Grund wird hier eine etwas vereinfachte praktikablere Notation eingeführt. Jedes Start-, End- und Abbruchkriterium K_Φ beim PLB wird folgende Form haben:

$$K_\Phi = \{(w1, w2) \in \left(\bigtimes_{p \in PP_{\Phi S} \cup PP_{\Phi E} \cup PP_{\Phi O}} SI_p \right)^2 | f(w1, w2) = wahr\}$$

Dabei ist $f(w1, w2)$ eine Boolsche Funktion, die mit Hilfe von Vergleichs-, Und- und Oder-Operationen auf den Signalwerten der in $w1$ und $w2$ enthaltenen Ports entweder zu $wahr$ oder $falsch$ auswertet. Der Wert eines speziellen Ports p wird dabei mit Hilfe der Projektionsfunktion $proj_p$ aus $w1$ oder $w2$ ermittelt. Für den PLB ist also ein Kriterium eindeutig bestimmt, wenn man festlegt, um welches Kriterium es sich handelt (Start, Ende oder Abbruch), für welche Phase das Kriterium ist und wenn man das entsprechende $f(w1, w2)$ benennt.

Zur weiteren Straffung der Notation wird anstelle von $proj_p(w1)$ nur p_1 und anstelle von $proj_p(w2)$ nur p_2 gesetzt. Darüber hinaus ist der Wert eines Ports genau dann $wahr$, wenn er ungleich Null ist. Die Negation dieser Aussage ist mit ! gegeben. Der Wechsel eines Wertes von Null auf ungleich Null sei mit $\uparrow$, der Wechsel von ungleich Null auf Null mit $\downarrow$ und ein Signalwechsel mit $\updownarrow$ gegeben. So kann zum Beispiel ein Startkriterium für eine Buphase FOO, die startet, wenn ein Signal $sig1$ von Null auf Eins wechselt oder wenn $sig2$ aktuell den Wert 42 hat oder wenn $sig3$ seinen Wert wechselt oder wenn $sig4$ im letzten Takt Eins und $sig5$ im aktuellen Takt Null ist, geschrieben werden als:

$S_{FOO} : (\uparrow sig1) \vee (sig2_2 = 42) \vee (\updownarrow sig3) \vee (sig4_1 \wedge !sig5_2))$

und steht dabei für

$$\begin{aligned} S_{FOO} = \{(w1, w2) \in & \left(\bigtimes_{p \in PP_{FOOS} \cup PP_{FOOE} \cup PP_{FOOO}} SI_p \right)^2 | \\ & (proj_{sig1}(w1) = 0 \wedge proj_{sig1}(w2) \neq 1) \vee \\ & (proj_{sig2}(w2) = 42) \vee \\ & (proj_{sig3}(w1) \neq proj_{sig3}(w2)) \vee \\ & (proj_{sig4}(w1) \neq 0 \wedge proj_{sig5}(w2) = 0)\} \end{aligned}$$

Diese Notation erhebt keinen Anspruch auf Vollständigkeit. Sie wird einzig zur Straffung für die Kriterien des PLB definiert. Sie illustriert jedoch, wie der Formalismus der Busphasenbeschreibung etwas praktikabler gemacht werden kann.

Manche Phasen benötigen zur Bestimmung eines Phasenrandes Informationen über den aktuellen Kommunikationszustand, der nicht direkt aus dem Zustand des Interface ablesbar ist, also weit in der Vergangenheit liegt. Für den AHB wurde dies in Anhang I mit den Konstrukten LR (siehe Seite 240) bzw. LRO (siehe Seite 247) gelöst. Eine ähnliche Konstruktion ist auch für den PLB denkbar, jedoch fügt sich diese nicht gut in die vereinfachte Notation ein. Aus diesem Grund wird hier eine andere Möglichkeit beschrieben.

In ein Interface wird (nur im Gedankenmodell) zusätzliche Logik integriert, die das Auftreten spezieller Ereignisse speichert und als neue Signale zur Verfügung stellt. Dann können diese Signale bei der Berechnung der Kriterien verwendet werden. In der Realität existieren diese Signale nicht, wie kann also ein Modellentwickler aus der ihm zur Verfügung stehenden Information dann die Phasenränder korrekt bestimmen? Die erzeugten Signale sollen Informationen entsprechen, die im Rahmen einer PLB-Kommunikation ohnehin von den beteiligten Modulen gespeichert werden müssen. Dann hat der Designer die Information, die er benötigt. Existieren Informationen nicht, weil das Modul ein gegebenenfalls optionales Feature nicht nutzt, so sollen diese Signale als Null angenommen werden.

```
1  //Folgende 6 Zusatzsignale werden im Master-Interface benoetigt
2  wire finalWrTranfer;//true, wenn letzes Wort im Transfer ist
3  reg ongoingWrs [1:0]; //Anzahl zurzeit laufender Writes
4  reg lastWrWasBurst; //true wenn letzter akzeptierter Write-Request ein Burst war
5  wire finalRdTranfer;//true, wenn letzes Wort im Transfer ist
6  reg ongoingRds [1:0]; //Anzahl zurzeit laufender Reads
7  reg lastRdWasBurst; //true wenn letzter akzeptierter Read-Request ein Burst war
8
9  always @(posedge PLB_clk) begin
10   //bei akzeptiertem Write-Request und gleichzetig endendem Write (-Burst)
11   // gibt es keine Aenderung in ongoingWrs
12   if (Mn_request && PLB_AAck && !Mn_RNW && finalWrTranfer && PLB_wDAck)
13     ongoingWrs <= ongoingWrs;
14   else
15   begin
16     //bei akzeptiertem Write-Request erhoeht sich die Anzahl laufender Writes
17     if (Mn_request && PLB_AAck && !Mn_RNW) ongoingWrs <= ongoingWrs +1;
18     else
19     //bei beendetem Write(-Burst) verringert sich die Anzahl laufender Writes
20     if (finalWrTranfer && PLB_wDAck) ongoingWrs <= ongoingWrs-1;
21   end
22
23   //bei akzeptiertem Write-Request, speichere burst flag von Mn_size
24   if (Mn_request && PLB_AAck && !Mn_RNW) lastWrWasBurst <=Mn_size[0];
25
26   //bei akzeptiertem Read-Request und gleichzetig endendem Read (-Burst)
27   // gibt es keine Aenderung in ongoingRds
28   if (Mn_request && PLB_AAck && Mn_RNW && finalRdTranfer && PLB_rDAck)
29     ongoingRds <= ongoingRds;
30   else
31   begin
32     //bei akzeptiertem Read-Request erhoeht sich die Anzahl laufender Reads
33     if (Mn_request && PLB_AAck && Mn_RNW) ongoingRds <= ongoingRds +1;
34     else
35     //bei beendetem Read(-Burst) verringert sich die Anzahl laufender Reads
36     if (finalRdTranfer && PLB_rDAck) ongoingRds <= ongoingRds-1;
37   end
38
39   //bei akzeptiertem Read-Request, speichere burst flag von Mn_size
40   if (Mn_request && PLB_AAck && Mn_RNW) lastRdWasBurst <= Mn_size[0];
41 end
```

Listing 6.2: zusätzliche Kommunikationszustandsinformation für das PLB-Master-Interfaces

Für den PLB zeigen Listings 6.2 und 6.3 die virtuell hinzugefügten Signale, erklären diese und zeigen ihre Berechnungen. Keine Berechnungen werden für `final[Rd|Wr]Transfer` in Listing 6.2 und `Sl_addressed` in Listing 6.3 angegeben. Der letzte Transfer eines Bursts

wird daran erkannt, dass [PLB|Mn]_rBurst bzw. [PLB|Mn]_wBurst auf Null sind. Der letzte Transfer einer Cacheline ergibt sich aus dem Mitzählen der Transfers, und bei Einzelwortransfers ist der erste auch der letzte Transfer. Dafür den Code anzugeben, würde den gezeigten Code unnötig aufblähen. Desweiteren legt jeder Slave selbst fest, welche Adressen für ihn gültig sind, wann also `Sl_addressed` auf Eins gesetzt ist, und somit wird dafür auch keine gesonderte Berechnung angegeben.

```
1  //Folgende 7 Zusatzsignale werden im Slave-Interface benoetigt
2  wire Sl_addressed; //true wenn der jeweilige Slave addressiert ist
3  wire finalWrTranfer;//true, wenn letzes Wort im Transfer ist
4  wire finalRdTranfer;//true, wenn letzes Wort im Transfer ist
5  reg secWr; //true, wenn secondary Write-Request akzeptiert und noch nicht promoted wurde
6  reg lastSecWrWasBurst; //true wenn letzter akzeptierter secondary Write-Request ein Burst war
7  reg secRd; //true, wenn secondary Read-Request akzeptiert und noch nicht promoted wurde
8  reg lastSecRdWasBurst; //true wenn letzter akzeptierter secondary Read-Request ein Burst war
9
10 always @(posedge PLB_clk) begin
11   //akzeptierter secondary Write-Request (merke ob Burst)
12   if (PLB_SAValid && Sl_AAck && !PLB_RNW) begin
13     secWr<=!PLB_wPrim;  //SAValid plus gleichzeitiges Prim ist wie PAValid
14     lastSecWrWasBurst <= PLB_size[0];
15   end
16   else
17   if (PLB_wPrim) secWr<=0; //Promotion
18
19   //akzeptierter secondary Read-Request
20   if (PLB_SAValid && Sl_AAck && PLB_RNW) begin
21     secRd<=!PLB_rPrim;  //SAValid plus gleichzeitiges Prim ist wie PAValid
22     lastSecRdWasBurst <= PLB_size[0];
23   end
24   else
25   if (PLB_rPrim) secRd<=0; //Promotion
26 end
```

Listing 6.3: zusätzliche Kommunikationszustandsinformation für ein PLB-Slave-Interfaces

REQ Die REQ-Phase im Master-Interface entspricht dem Address-Cycle des PLB (siehe [IBM-04]). Sie startet, wenn Mn_abort keinesfalls Eins ist, Mn_request auf Eins wechselt (Start eines neuen Requests), wenn Mn_request in zwei aufeinanderfolgenden Takten Eins ist und im ersten dieser Takte PLB_AAck, PLB_tout, PLB_rearb oder Mn_abort auf Eins war (Start eines Requests direkt nach Ende oder Abbruch eines Requests) oder wenn Mn_request in zwei aufeinanderfolgenden Takten Eins ist und im zweiten Takt Mn_priority einen anderen Wert hat als im ersten (Neustart bei Prioritätsänderung).

$$
\begin{aligned}
S_{REQ}: \quad & (!Mn\_abort_2 \wedge \uparrow Mn\_request) \\
& \vee \\
& [!Mn\_abort_2 \wedge Mn\_request_2 \wedge \\
& \quad (PLB\_rearb_1 \vee PLB\_AAck_1 \vee Mn\_abort_1 \vee PLB\_tout_1 \vee \updownarrow Mn\_priority)]
\end{aligned}
$$

Die Phase endet, wenn Mn_request gesetzt ist und PLB_rearb (Slave kann zurzeit den Request nicht akzeptieren), PLB_AAck (Slave akzeptiert den Request) oder PLB_tout (Time-Out) gesetzt sind und der Master die Phase nicht abbricht.

$$E_{REQ}: \quad Mn\_request_2 \wedge !Mn\_abort_2 \wedge (PLB\_rearb_2 \vee PLB\_AAck_2 \vee PLB\_tout_2)$$

Die Phase bricht ab, wenn der Master zu Mn_request Mn_abort auf Eins setzt ist.

$$A_{REQ}: \quad Mn\_request_2 \wedge Mn\_abort_2$$

REQ$_{SL}$ Die REQ$_{SL}$-Phase im Slave-Interface entspricht dem Primary- oder dem Secondary-Address-Cycle des PLB (siehe [IBM-04]). Sie verhält sich grundsätzlich wie die REQ-Phase des Masters unter Umbenennung der Signale (ersetzte Mn_ durch PLB_ und PLB_ durch Sl_). Unterschiede sind, dass es slaveseitig kein PLB_request-Signal gibt, sondern zwei Signale PLB_PAValid und PLB_SAValid, die neben der Tatsache, dass ein Addresszyklus läuft, diesen als „primary“ oder „secondary“ markieren und dass der Slave keine TimeOut-Information (kein PLB_tout-Signal) erhält.

Man beachte hier, dass Sl_addressed in die Bestimmung zum Start und Ende der Phase herangezogen wird. Dies bedeutet, dass die REQ$_{SL}$-Phase immer nur im adressierten Slave abläuft. Damit ein PLB-Modell, welches die Phase startet, dies tun kann, muss es also über Adressinformationen verfügen, die in der Realität nur der Slave besitzt. Dies bedeutet einen geringen Mehraufwand bei der Integration des Modells (eine Adress-Map muss erzeugt und an das Modell übergeben werden), erlaubt aber eine deutlich effizientere Simulation, da nicht immer alle Slaves angesprochen werden müssen.

$$\begin{aligned} S_{REQ_{SL}}: \quad & (Sl\_addressed_2 \wedge !PLB\_abort_2 \wedge (\uparrow PLB\_PAValid \vee \uparrow PLB\_SAValid)) \\ & \vee \\ & [Sl\_addressed_2 \wedge !PLB\_abort_2 \wedge (PLB\_PAValid_2 \vee PLB\_SAValid_2) \\ & \quad \wedge (Sl\_rearb_1 \vee Sl\_AAck_1 \vee PLB\_abort_1 \vee \updownarrow PLB\_priority)] \end{aligned}$$

$$\begin{aligned} E_{REQ_{SL}}: \quad & (PLB\_PAValid_2 \vee PLB\_SAValid_2) \wedge !PLB\_abort_2 \wedge \\ & \quad (PLB\_rearb_2 \vee PLB\_AAck_2) \end{aligned}$$

$$A_{REQ_{SL}}: \quad Sl\_addressed_2 \wedge (PLB\_PAValid_2 \vee PLB\_SAValid_2) \wedge PLB\_abort_2$$

RESP$_{(SL)}$ Die RESP- bzw. die RESP$_{SL}$-Phase markiert das Liefern eines Lesedatums vom Slave zum Master. Ein solches Lesedatum ist nur genau einen Takt lang gültig, die Einzeltaktphase endet und startet also wenn PLB_rDAck bzw. Sl_rDAck auf Eins ist. Stellvertretend werden hier nur die Kriterien der RESP-Phase gezeigt. Die Kriterien der RESP$_{SL}$-Phase ergeben durch triviale Signalumbenennungen

$$S_{RESP}: \quad PLB\_rDAck_2 \; ; E_{RESP} = S_{RESP}.$$

RBT$_{(SL)}$ Die RBT (Resp Burst Terminate)- bzw. die RBT$_{SL}$-Phase markiert den Takt, in dem der Slave den Master zur Burst-Terminierung aufruft. Dies geschieht durch das Setzen von PLB_rBTerm bzw. Sl_rBTerm für einen Takt. Die Kriterien der RBT-Phase sind genannt, die Kriterien für RBT$_{SL}$-Phase ergeben durch triviale Signalumbenennungen

$$S_{RBT}: \quad PLB\_rBTerm_2 \; ; E_{RBT} = S_{RBT}.$$

RBC$_{(SL)}$ Die RBC (Resp Burst Cancel)- bzw. die RBC$_{SL}$-Phase markiert den Takt, in dem der Master einen Burst in für den Slave nicht vorhersehbarer Weise beendet. Der Verlauf eines Bursts wird im PLB mit Hilfe der *_rBurst-Signale gesteuert. Für Einzelwort-, Cacheline- und Fixed-Length-Burst-Transfers ist der Verlauf für den Slave bei Einhaltung des PLB-Protokolls vorhersagbar. Da im Rahmen der Performance-Evaluation von der Einhaltung des PLB-Protokolls ausgegangen werden kann, genügt es also, nur die nicht vorhersehbaren Abbrüche zu markieren. Diese sind das Abbrechen eines Bursts, ohne dass der Slave vorher die Terminierung veranlasst hat, und der primäre oder sekundäre Start eines Bursts der Länge Eins (nicht zu Verwechseln mit einem Einzelworttransfer). Zu den Phasenstart und -endports gehören nur `Mn_rBurst` bzw. `PLB_rBurst`. Somit sind alle weiteren, in den unten stehenden Berechnungen gelisteten Ports Teil der Obersevierungsports $PP_{RBC_{(SL)}O}$ dieser Phase.

$$
\begin{aligned}
S_{RBC}: \quad & (\downarrow Mn\_rBurst \wedge !PLB\_rBTerm_1) \\
& \vee \\
& (Mn\_request_1 \wedge PLB\_AAck_1 \wedge Mn\_RNW_1 \wedge Mn\_size[0]_1 \wedge !Mn\_rBurst_2 \\
& \quad \wedge !ongoingRds_1) \\
& \vee \\
& (PLB\_rDAck_1 \wedge finalRdTranser_1 \wedge ongoingRds_2 > 0 \\
& \quad \wedge lastRdWasBurst_2 \wedge !Mn\_rBurst_2)
\end{aligned}
$$

$$E_{RBC} = S_{RBC}$$

$$
\begin{aligned}
S_{RBC_{SL}}: \quad & (\downarrow PLB\_rBurst \wedge !Sl\_rBTerm_1 \wedge Sl\_rDAck_1) \\
& \vee \\
& ((PLB\_PAValid_1 \vee (PLB\_SAValid_1 \wedge PLB\_rPrim_1)) \wedge Sl\_AAck_1 \\
& \quad \wedge PLB\_RNW_1 \wedge PLB\_size[0]_1 \wedge !PLB\_rBurst_2) \\
& \vee \\
& (PLB\_rPrim_1 \wedge secRd_1 \wedge lastSecRdWasBurst_1 \wedge !Mn\_rBurst_2)
\end{aligned}
$$

$$E_{RBC_{SL}} = S_{RBC_{SL}}$$

DATA$_{(SL)}$ Die DATA- bzw. die DATA$_{SL}$-Phase markiert das Überragen eines Schreibdatums vom Master zum Slave. Das Schreibdatum wird entweder zusammen mit dem Request eines Masters gültig, wenn der Master nicht bereits in einer Schreibdatenphase ist, oder es wird direkt nach dem Ende der vorhergehenden Datenphase gültig, wenn zurzeit ein Request läuft, wenn der akuelle Burst noch nicht beendet ist oder wenn ein sekundärer Schreibtransfer anhängig ist. Alle zur Bestimmung des Starts von DATA$_{(SL)}$ herangezogenen Signale ausser *_wDBus, *_wDAck und *_MwErr sind Teil von $PP_{DATA_{(SL)}O}$.

$$
\begin{aligned}
S_{DATA}: \quad & (S_{REQ} \wedge !Mn\_RNW_2 \wedge !ongoingWrs) \\
& \vee \\
& [PLB\_wDAck_1 \wedge \\
& (!finalWrTransfer_1 \vee (Mn\_request_2 \wedge !Mn\_RNW_2) \vee (ongoingWrs_2 > 0))]
\end{aligned}
$$

Die DATA-Phase endet mit dem Setzen von *_wDAck.

$E_{DATA} = PLB\_wDAck$

Für die DATA$_{SL}$-Phase heisst das dann

$$\begin{aligned} S_{DATA_{SL}}: \quad & (\uparrow PLB\_PAValid \wedge !PLB\_RNW_2 \wedge Sl\_addressed_2) \\ & \vee \\ & [(Sl\_wDAck_1 \wedge !finalWrTransfer_1) \vee (PLB\_wPrim_1 \wedge secWr_1) \\ & \quad \vee (PLB\_wPrim_1 \wedge PLB\_SAValid_1 \wedge Sl\_addressed_1)] \end{aligned}$$

$E_{DATA_{SL}} = Sl\_wDAck$

DBT$_{(SL)}$ Die DBT (Data Burst Terminate)- bzw. die DBT$_{SL}$-Phase markiert den Takt, in dem der Slave den Master zur Burstterminierung aufruft. Dies geschieht durch das Setzen von PLB_wBTerm bzw. Sl_wBTerm für einen Takt. Die Kriterien der DBT-Phase sind genannt, die Kriterien für DBT$_{SL}$-Phase ergeben durch triviale Signalumbenennungen.

$S_{DBT}: \quad PLB\_wBTerm_2 \; ; E_{DBT} = S_{DBT}$

DBC$_{(SL)}$ Die DBC (Data Burst Cancel)- bzw. die DBC$_{SL}$-Phase markiert den Takt, in dem der Master einen Burst in für den Slave nicht vorhersehbarer Weise beendet (vergleiche dazu die Erläuterungen zur Phase RBC). Zu den Phasenstartports und -endports gehören nur `Mn_wBurst` bzw. `PLB_wBurst`. Somit sind alle weiteren in den untenstehenden Berechnungen gelisteten Ports Teil von $PP_{DBC_{(SL)}O}$.

$$\begin{aligned} S_{DBC}: \quad & (\downarrow Mn\_wBurst \wedge !PLB\_wBTerm_1) \\ & \vee \\ & (\uparrow Mn\_request \wedge !Mn\_RNW_2 \wedge Mn\_size[0]_2 \wedge !Mn\_wBurst_2 \wedge !ongoingWrs_2) \\ & \vee \\ & (PLB\_wDAck_1 \wedge finalWrTransfer_1 \wedge !Mn\_wBurst_2 \\ & \quad \wedge ((ongoingWrs_2 > 0 \wedge lastWrWasBurst_2) \\ & \quad \vee (Mn\_request_2 \wedge !Mn\_RNW_2 \wedge Mn\_size[0]_2))) \end{aligned}$$

$E_{DBC} = S_{DBC}$

$$\begin{aligned} S_{DBC_{SL}}: \quad & (\downarrow PLB\_wBurst \wedge !Sl\_wBTerm_1 \wedge Sl\_wDAck_1) \\ & \vee \\ & (\uparrow PLB\_PAValid \wedge Sl\_addressed_2 \wedge !PLB\_RNW_2 \wedge PLB\_size[0]_2 \\ & \quad \wedge !PLB\_wBurst_2) \\ & \vee \\ & (PLB\_wPrim_1 \wedge secWr_1 \wedge lastSecWrWasBurst_1 \wedge !Mn\_wBurst_2) \\ & \vee \\ & (PLB\_wPrim_1 \wedge Sl\_addressed_1 \wedge PLB\_SAValid_1 \wedge Sl\_AAck_1 \\ & \quad \wedge PLB\_size[0]_1 \wedge !Mn\_wBurst_2) \end{aligned}$$

$E_{DBC_{SL}} = S_{DBC_{SL}}$

WAIT Diese Phase markiert den Takt, in dem der Slave die Time-Out-Zählung für einen primären Request deaktiviert. Nach dem Setzen für einen Request bleibt das Signal laut

PLB-Spezifikation gesetzt bis der nächste Request beginnt. Somit reicht es aus, nur den Takt zu markieren, in dem das Signal erstmals für einen Request gültig ist. Zu den Phasenstartports und -endports gehört nur `Sl_wait`. Somit sind alle weiteren, in den unten stehenden Berechnungen gelisteten Ports Teil von PP_{WAITO}.

$$S_{WAIT}: \; Sl\_addressed_2 \wedge PLB\_PAValid_2 \wedge \\ [\uparrow Sl\_wait \vee ((Sl\_AAck_1 \vee Sl\_rearb_1 \vee PLB\_abort_1) \wedge Sl\_wait_2)]$$

$$E_{WAIT} = S_{WAIT}$$

RCOMP Diese Phase markiert den Takt, in dem der Slave den PLB anweist, den Read-Bus freizugeben.

$$S_{RCOMP}: \; Sl\_rComp \; ; \; E_{RCOMP} = S_{RCOMP}$$

DCOMP Diese Phase markiert den Takt, in dem der Slave den PLB anweist, den Write-Bus freizugeben.

$$S_{DCOMP}: \; Sl\_wComp \; ; \; E_{DCOMP} = S_{DCOMP}$$

RPRIM Diese Phase markiert den Takt, in dem der PLB dem Slave mitteilt, dass sein sekundärer Read-Transfer jetzt primär ist. Man beachte, dass *secRd* nur in genau dem Slave-Interface auf Eins ist, an den der ursprüngliche Request gerichtet war.

$$S_{RPRIM}: \; PLB\_rPrim_2 \wedge secRd_2 \; ; \; E_{RPRIM} = S_{RPRIM}$$

DPRIM Diese Phase markiert den Takt, in dem der PLB dem Slave mitteilt, dass sein sekundärer Write-Transfer jetzt primär ist. Man beachte, dass $secWr$ nur in genau dem Slave-Interface auf Eins ist, an den der ursprüngliche Request gerichtet war.

$$S_{DPRIM}: \; PLB\_wPrim_2 \wedge secWr_2 \; ; \; E_{DPRIM} = S_{DPRIM}$$

IRQ$_{(SL)}$ Diese Phase startet in einem Interface, sobald *_MIRQ ungleich Null ist, startet dann erneut mit jeder Änderung und bricht ab, wenn das Signal wieder Null ist. So wird jede Änderung von *_MIRQ kommuniziert. Die Kriterien der IRQ-Phase sind genannt, die Kriterien für IRQ$_{SL}$-Phase ergeben durch triviale Signalumbenennungen.

$$S_{IRQ}: \; \updownarrow PLB\_MIRQ \wedge PLB\_MIRQ_2 \; ; \; A_{IRQ}: \; \downarrow PLB\_MIRQ$$

BUSY$_{(SL)}$ Analog zur IRQ-Phase, startet diese Phase in einem Interface, sobald *_MBusy ungleich Null ist, startet dann erneut mit jeder Änderung und bricht ab, wenn das Signal wieder Null ist. Die Kriterien der BUSY-Phase sind genannt, die Kriterien für BUSY$_{SL}$-Phase ergeben durch triviale Signalumbenennungen.

$$S_{BUSY}: \; \updownarrow PLB\_MBusy \wedge PLB\_MBusy_2 \; ; \; A_{BUSY}: \; \downarrow PLB\_MBusy$$

DPND$_{(SL)}$ Diese Phase startet in einem Interface, sobald PLB_wPdRq gesetzt ist oder wenn sich bei gesetztem PLB_wPdRq PLB_wPdPri ändert. Sie endet sobald PLB_wPdRq wieder auf Null ist. Es wird also stets angezeigt, ob auf dem PLB ein Write-Request anhängig

ist und mit welcher Priorität. Die Kriterien der DPND-Phase sind genannt, die Kriterien für die DPND_{SL}-Phase sind identisch.

$S_{DPND}: \uparrow PLB\_wPdRq \vee (\updownarrow PLB\_wPdPri \wedge PLB\_wPdRq_2)$

$A_{DPND}: \downarrow PLB\_wPdRq$;

RPND$_{(SL)}$ Die RPND-Phasen verhalten sich analog zu den DPND-Phasen. Die Kriterien der RPND-Phase sind genannt, die Kriterien für die RPND_{SL}-Phase sind identisch.

$S_{RPND}: \uparrow PLB\_rPdRq \vee (\updownarrow PLB\_rPdPri \wedge PLB\_rPdRq_2)$

$A_{RPND}: \downarrow PLB\_rPdRq$;

Damit sind die Phasen vollständig erfasst, und das GP- und TLM-Phase-Mapping kann stattfinden. Das Ergebnis des GP-Mapping wurde bereits in Tabelle 6.1 gezeigt. Die Entstehung des Mapping wird nicht im Einzelnen erläutert (dazu sei auf Anhang I verwiesen). Es sei an dieser Stelle noch einmal darauf hingewiesen, dass ein Port dann nicht im GP-Mapping behandelt werden muss, wenn sein Wert aus der An- bzw. Abwesenheit von Phasen abgeleitet werden kann. Er kann dann implizit modelliert werden.

Das Ergebnis des TLM-Phase-Mappings wird in Tabelle 6.4 gezeigt. Für Einzeltaktphasen sind Start und Ende in einer Zeile dargestellt, da diese immer im gleichen Takt auftreten. Die Zusammenlegung der Phasen, die sowohl mit als auch ohne $_{SL}$ Zusatz existieren, ergibt sich für alle Phasen ausser $\text{REQ}_{(SL)}$ bereits im TLM-Phase-Mapping, da diese Phasen die gleichen GP-Elemente verwenden, die gleichen impliziten Modellierungen treffen und sich innerhalb eines Interfaces zeitlich nicht überlappen (siehe Abschnitt 4.3.2). Für REQ und REQ_{SL} gilt dies nicht, da sich diese beiden Phasen in den implizit modellierten Signalen unterscheiden (Mn_request). Die Zusammenlegung dieser beiden Phasen ergibt sich aus der TLM-Phasenreduktion (siehe Abschnitt 4.3.4). Beide Phasen können nur auf unterschiedlichen Punkt-zu-Punkt-Verbindungen auftreten, wenn man die Information, ob eine Punkt-zu-Punkt-Verbindung eine Master→Bus- oder eine Bus→Slave-Verbindung ist, als vorhanden ansieht[5]. Die Phasen widersprechen sich nicht in den impliziten Signalen und haben gemeinsame GP-Grundelemente und GP-Erweiterungen. Die TLM-Phasenreduktion kann die entsprechenden `BEGIN_`-, `ABORT_`- und `END_`-TLM-Phasen-Paare der beiden Phasen folglich zu je einer TLM-Phase zusammenlegen.

[5]Da ein Modul stets weiß ob es Master oder Slave einer Verbidnung ist, ist dies durchaus sinnvoll.

TLM-Phase	Repräsentant für	implizit modellierte Signale	GP-Grundelemente
BEGIN_REQ	Start: REQ, REQ_{SL}	*_abort=0, Mn_request=1 (nur Master)	m_command, m_byte_enable[_length], m_address
ABORT_REQ	Abbruch: REQ, REQ_{SL}	*_abort=1, Mn_request=1 (nur Master)	m_command, m_byte_enable[_length], m_address
END_REQ	Ende: REQ, REQ_{SL}	*_abort=0, Mn_request=1 (nur Master)	keine
[BEGIN|END]_RESP	Start,Ende: RESP, $RESP_{SL}$	*_rDAck=1	m_data[_length], m_response_status
[BEGIN|END]_RBT	Start,Ende: RBT, RBT_{SL}	*_rBTerm=1	keine
[BEGIN|END]_RBC	Start,Ende: RBC, RBC_{SL}	*_rBurst=0	keine
BEGIN_DATA	Start: DATA, $DATA_{SL}$	keine	m_data[_length]
END_DATA	Ende: DATA, $DATA_{SL}$	*_wDAck=1	m_response_status
[BEGIN|END]_DBT	Start,Ende: DBT, DBT_{SL}	*_wBTerm=1	keine
[BEGIN|END]_DBC	Start,Ende: DBC, DBC_{SL}	*_wBurst=0	keine
[BEGIN|END]_WAIT	Start,Ende: WAIT	Sl_wait=1	keine
[BEGIN|END]_RCOMP	Start,Ende: RCOMP	Sl_rComp=1	keine
[BEGIN|END]_DCOMP	Start,Ende: DCOMP	Sl_wComp=1	keine
[BEGIN|END]_RPRIM	Start,Ende: RPRIM	PLB_rPrim=1	keine
[BEGIN|END]_DPRIM	Start,Ende: DPRIM	PLB_wPrim=1	keine
BEGIN_IRQ	Start: IRQ, IRQ_{SL}	keine	keine
ABORT_IRQ	Abbruch: IRQ, IRQ_{SL}	*_MIRQ=0	keine
BEGIN_BUSY	Start: BUSY, $BUSY_{SL}$	keine	keine
ABORT_BUSY	Abbruch: BUSY, $BUSY_{SL}$	*_MBusy=0	keine
BEGIN_DPND	Start: DPND, $DPND_{SL}$	PLB_wPdRq=1	keine
ABORT_DPND	Abbruch: DPND, $DPND_{SL}$	PLB_wPdRq=0	keine
BEGIN_RPND	Start: RPND, $RPND_{SL}$	PLB_rPdRq=1	keine
ABORT_RPND	Abbruch: RPND, $RPND_{SL}$	PLB_rPdRq=0	keine

Tabelle 6.4.: Übersicht über das Ergebnis des TLM-Phase-Mappings für PLB-Master und -Slave

Aufbauend auf dem GP- und TLM-Phase-Mapping werden die Modifiabilities der GP-Grundelemente und GP-Erweiterungen festgelegt. Die Festlegung der Phasenassoziationen geschieht gemäß Abschnitt 4.7.1 vor der TLM-Phasenreduktion, also vor der Zusammenlegung von REQ und REQ_{SL}. Nach der Zusammenlegung von REQ und REQ_{SL} wird jedes Auftreten von `[BEGIN|ABORT|END_REQSL]` in einer Phasenassoziation durch `[BEGIN|ABORT|END_REQ]` ersetzt und dadurch entstehende Doppelnennungen wurden dann entfernt. Das Endergebnis ist in Tabelle 6.5 gezeigt.

GP-Element	assoziierte Phasen	Intervall	Erklärung
GP-Grundelemente			
`m_command`	`BEGIN_REQ`, `ABORT_REQ`	e2e	Transferrichtung bleibt immer gleich.
`m_address`	`BEGIN_REQ`, `ABORT_REQ`	x2x	Auf einer Verbindung konstant für eine Transaktion, kann sich aber in Busbrücken ändern.
`*m_byte_enable`	`BEGIN_REQ`, `ABORT_REQ`	e2e	Laut PLB-Spezifikation gilt eine Byte-Enable-Maske für alle Worte des Bursts. Kann sich auch in Brücken nicht ändern.
`*m_data`	`BEGIN_RESP`, `BEGIN_DATA`	byte-weise e2e	Einmal gesetzt bleiben Daten gültig. Ein Byte x des Datenfeldes ist mit dem n-ten Auftreten einer Phase aus der Phasenassoziation verbunden, wobei $n = \lfloor \frac{x}{size} \rfloor + 1$ gilt. Dabei ist $size = 4$, bei Cache-Line-Transfers, $size = data\_bus\_in\_bytes$ bei Einzeltransfers und $size = 2^{Mn\_size[1:3]}$ bei Bursts.
`m_response_status`	`BEGIN_RESP`, `END_DATA`	p2p	Jeder Einzeltransfer in einem Burst kann eine eigene Response haben (Error oder nicht). Somit liegt eine zeitliche Varianz vor.
GP-Erweiterungen			
`is_prim`, `is_sec`, `priority`, `is_locked`, `trans_size`, `trans_type`, `mst_size`, `attribs`, `lck_err_reg`, `mst_id`,	`BEGIN_REQ`, `ABORT_REQ`	x2x	Die Transferqualifizierer sind für eine Transaktion konstant, können sich aber innerhalb von Busbrücken ändern.

endtype, slv_size	END_REQ	p2p	Der Antworttyp kann sich in Brücken ändern (ein Time-Out auf einem PLB, kann als rearbitrate-Antwort auf einen anderen Bus gelangen), bleibt aber für eine Transaktion konstant. Auch die Information über die Datenbreite eines Slaves kann sich in Busbrücken ändern. Diese Werte sind also prinzipiell x2x-änderbar. Jedoch kann eine Busbrücke ein END_REQ an ihren Master-Bus senden, bevor das END_REQ vom Slave-Bus Empfangen wurde. Trotz des x2x-Ändderungsintervalls müssen die Werte also wie p2p-änderbar benutzt werden (siehe Abschnitt 4.7.4, Seite 83).
rdAddr	BEGIN_RESP	p2p	Jede Cache-Line-Read-Response hat einen eigenen Wert für diese Erweiterung, es liegt also zeitliche Veränderbarkeit vor.
irq_set	BEGIN_IRQ	p2p	Die IRQ-Maske kann sich mehrfach im Verlauf ändern (bei Neustart der Phase aufgrund einer IRQ-Maskenänderung). Es liegt also zeitliche Veränderbarkeit vor.
bsy_set	BEGIN_BSY	p2p	Der Busy-Zustand kann sich mehrfach im Verlauf ändern (bei Neustart der Phase aufgrund einer Busy-Zustandsänderung). Es liegt also zeitliche Veränderbarkeit vor.
d_pd_prit	BEGIN_DPND	p2p	Die Priorität eines anhängigen Schreib-Requests kann sich mehrfach im Verlauf ändern (bei Neustart der Phase aufgrund einer Prioritätsänderung). Es liegt also zeitliche Veränderbarkeit vor.
r_pd_prit	BEGIN_RPND	p2p	Die Priorität eines anhängigen Lese-Requests kann sich mehrfach im Verlauf ändern (bei Neustart der Phase aufgrund einer Prioritätsänderung). Es liegt also zeitliche Veränderbarkeit vor.

Tabelle 6.5.: Modifiabilities der GP-Elemente für den PLB

Das initiale und anschließende mehrfache GP- und TLM-Phase-Mapping jeder möglichen Kombination von optionalen Signalen[6] führt dann zu den in Tabelle 6.6 gezeigten Sätzen von verhandelbaren und unverhandelbaren GP-Erweiterungen und TLM-Phasen. Sockets für Master-Interfaces (also Initiator-Sockets von Master-Modellen und Target-Sockets von PLB-Modellen) müssen die GP-Erweiterungen `is_prim`, `is_sec` und `master_id` und die TLM-Phasen `[BEGIN|END]_WAIT`, `[BEGIN|END]_RCOMP`, `[BEGIN|END]_DCOMP`, `[BEGIN|END]_RPRIM`, `[BEGIN|END]_DPRIM` grundsätzlich als „verboten“ markieren, da es im Master-Interface dafür keine Entsprechungen gibt (vergleiche Tabelle 6.1), während Sockets für das Slave-Interface (also Initiator-Sockets von PLB-Modellen und Target-Sockets von Slave-Modellen) diese GP-Erweiterungen und TLM-Phasen als „zwingend“ markieren müssen, da ihnen die Verwendung nicht frei steht.

	GP-Erweiterungen	TLM-Phasen
unverhandelbar	`end_type`, `bsy_set`, `d_pd_pri`, `r_pd_pri`	`[BEGIN|END]_REQ`, `[BEGIN|END]_RESP`, `[BEGIN|END]_DATA`, `[BEGIN|ABORT]_BSY`, `[BEGIN|ABORT]_DPND`, `[BEGIN|ABORT]_RPND`
verhandelbar	`is_prim`, `is_sec`, `priority`, `is_locked`, `trans_size`, `trans_type`, `mst_size`, `attribs`, `lck_err_reg`, `master_id`, `slv_size`, `rdAddr`, `irq_set`	`[BEGIN|END]_RCOMP`, `[BEGIN|END]_DCOMP`, `[BEGIN|END]_RPRIM`, `[BEGIN|END]_DPRIM`, `[BEGIN|END]_RBT`, `[BEGIN|END]_RBC`, `[BEGIN|END]_DBT`, `[BEGIN|END]_DBC`, `[BEGIN|END]_WAIT`, `[BEGIN|ABORT]_IRQ`

Tabelle 6.6.: Sätze verhandelbarer und nicht-verhandelbarer GP-Erweiterungen und TLM-Phasen für den PLB

Ansonsten ergeben sich die L1-Konfiguration für die verhandelbaren Sätze, abzüglich der bereits oben erwähnten GP-Erweiterungen und TLM-Phasen, nach einfachen Regeln:

1. Der Erfordernisgrad nicht explizit genannter Phasen oder Erweiterungen ist auf „verboten“ zu setzen.

2. Für einen Socket eines Moduls gilt:

2.1 Ergibt sich eine TLM-Phase oder eine GP-Erweiterung aus dem Mapping eines Signales, das vom Modul getrieben wird (bei Mastern alle Signale mit Präfix „Mn_“, bei Slaves alle Signale mit Präfix „Sl_“, beim PLB alle Signale mit Präfix „PLB_“), und ist dieses

[6] Die Namen dieser Signale sind in Tabelle 6.1 in eckige Klammern gesetzt.

Signal Bestandteil des Interfaces des modellierten Moduls, so ist der entsprechende Erfordernisgrad auf „zwingend“[7] zu setzen.

2.2 Ergibt sich eine TLM-Phase oder eine GP-Erweiterung aus dem Mapping eines Signales, das vom Modul empfangen wird (bei Mastern und Slaves alle Signale mit Präfix „PLB_“, beim PLB alle Signale mit Präfix „Mn_“ oder „Sl_“), so ist dieses Signal ein nicht-optionaler Bestandteil des Interfaces, und die Verwendung der TLM-Phase bzw. GP-Erweiterung ist unvermeidbar. In diesem Fall wird der Erfordernisgrad auf „optional“ gesetzt[8].

Dieses Vorgehen stellt sicher, dass Master und Slaves nicht direkt verbunden werden können (wegen `is_prim`, `is_sec`, etc.) und dass Master und Slaves nur an PLBs angeschlossen werden können, die mindestens alle Features unterstützen, die der Master bzw. Slaves auch verwenden will.

Zusätzlich werden noch folgende Regeln für die Verwendung des PLB-TLM-2.0-Interfaces festgelegt:

- Zur Signalisierung der BUSY-, IRQ- und *PND-Phasen sind eigene GPs zu nutzen, die vom Target (PLB- oder Slave-Modell) instanziiert werden müssen. Solch ein GP darf vom Sender auf mehrere Punkt-zu-Punkt-Verbindungen übermittelt werden (da z.B. der PLB allen Mastern die IRQ-Maske sendet), vom Empfänger aber nicht auf andere Verbindungen weitergeleitet werden. Sie dienen ausschließlich zur Signalisierung des Busy- oder IRQ-Zustandes auf den Punkt-zu-Punkt-Verbindungen, auf denen sie übertragen wurden.

- Eine Transaktion beginnt stets mit `BEGIN_REQ` und endet Master-seitig mit dem letzten `END_RESP`, `END_DATA` oder `ABORT_REQ`. Slave-seitig kann das Ende darüber hinaus auch mit `END_RCOMP` oder `END_DCOMP` zusammenfallen, wenn diese nach dem letzen `END_RESP` bzw. `END_DATA` auftreten. Der Slave muss in diesem Fall gemäß des TLM-2.0-Standards die Lebensdauer des GP mit Hilfe des Memory-Management des GP bis zum Ende der RCOMP- bzw. DCOMP-Phase sicherstellen.

- RBT, RBC, DBT, DBC, WAIT, RCOMP, DCOMP, RPRIM und DPRIM können logisch einzelnen Transaktionen zugeordnet werden und verwenden daher auch das gleiche GP wie die REQ-, DATA -und RESP-Phasen dieser Transaktion.

- Das Datenfeld muss vom Master vor dem Senden von `BEGIN_REQ` alloziert und die Länge des Datenfeldes im GP entsprechend gesetzt werden. Das Füllen des Feldes ist über dessen Modifiability geregelt. Im Falle eines Bursts undefinierter Länge muss der Master eine vordefinierte Maximalgröße allozieren[9].

[7] Das Modul ist Sender der GP-Erweiterung oder TLM-Phase und so kann es ihm nicht egal sein, ob die Erweiterung oder Phase verstanden wird.

[8] Das Modul ist Empfänger der optionalen GP-Erweiterung oder TLM-Phase und so ist es ihm egal ob die Erweiterung oder Phase benutzt wird. Ist sie nicht da, kann es den Defaultwert benutzen.

[9] Laut PLB-Spezifikation darf die Dauer eines Burst 1024-Takte nicht überschreiten. Die maximale Größe eines Bursts ist also die Datenbusbreite des Master multipliziert mit 1024.

- Das Byte-Enable-Feld muss vor dem Senden von `BEGIN_REQ` passend zur Datenbusbreite des Masters alloziert und die Länge des Byte-Enable-Feldes im GP entsprechend gesetzt werden (z.B. 8-Bytes bei einem 64-Bit-Master). Wird es nicht alloziert und die Länge auf Null gesetzt, werden alle Bytes des Datenfeldes als aktiv angenommen.

Die daraus resultierende Implementierung der GP-Erweiterungen, TLM-Phasen und Sockets ergibt sich analog zu dem im Anhang J gezeigten Vorgehen für den AHB.

6.3. Modellierung des Xilinx CoreConnect PLB v4.6

Zur detaillierten Evaluation des mit Hilfe der J-R-Simulation mit TLM-2.0 erzielbaren Speed-Ups wurde aus einem RTL-Model [XILI10] eines PLB v4.6 [IBM-04] manuell ein TLM-2.0-Modell für die taktgenaue J-R-Simulation zur Performance-Evaluation erstellt[10]. Als TLM-Interface wurde das in Abschnitt 6.2 definierte verwendet.

Zur Erstellung des Modells wurden die Spezifikationen von IBM [IBM-04] sowie die Produktbeschreibung von XILINX [XILI10] herangezogen. Zusätzlich wurde eine Referenz-Simulation in RTL mit Hilfe der XILINX-RTL-Implementierung aufgebaut, die aus einer Instanz des PLB sowie einer konfigurierbaren Anzahl von Mastern und Slaves besteht. Dabei sind die Master konfigurierbare Traffic-Generatoren und die Slaves Datensenken mit konfigurierbaren Latenzen. Nach einer Implementierung des TLM-Modells wurde eine der RTL-Referenz-Simulation entsprechende Simulation in TLM aufgebaut. Es wurde eine große Zahl von eingeschränkt zufällig erzeugten Abläufen verwendet, um zu verifizieren, dass die TLM-Simulation entsprechend der Inputabstraktion J und der Relevanzauswahl R, wie sie sich aus den Busphasendefinitionen ergeben, taktgenaue Simulationsergebnisse liefert.

6.3.1. Funktionsweise

Das TLM-Modell des PLB hat einen Multi-Targetsocket, an dem die Master angeschlossen werden, und einen Multi-Initiatorsocket, an dem die Slaves angeschlossen werden. Zeitannotationen werden mit Hilfe von Payload-Event-Queues (siehe [OSCI09a]$_i$) behandelt. Die entstandene Implementierung wird im Folgenden knapp beschrieben, weitere Details können der Implementierung entnommen werden [Günz10b]. Abbildung 6.6 illustriert die Arbeitsweise der Arbitrierung: Es gibt keinen aktiven Prozess. Das Modell wartet auf eingehende Kommunikation. Beim Empfang einer Request-Phase wird zuerst die Adresse zum gewünschten Index am Initiator-Socket dekodiert und das Paar aus diesem Index und dem Index des Targetsockets, von dem das GP empfangen wurde, werden mit Hilfe einer sog. Instance-Specific-Extension (siehe [OSCI09a]$_i$) an das GP annotiert. Anschließend entfallen jegliche Routing- oder Multiplexing-Aktivitäten, da das GP einfach zwischen diesen beiden Sockets ausgetauscht wird.

[10] Die Version 4.6 des PLB wurde verwendet, da für Version 4.7 [IBM-07] keine RTL-Implementierung zur Verfügung stand.

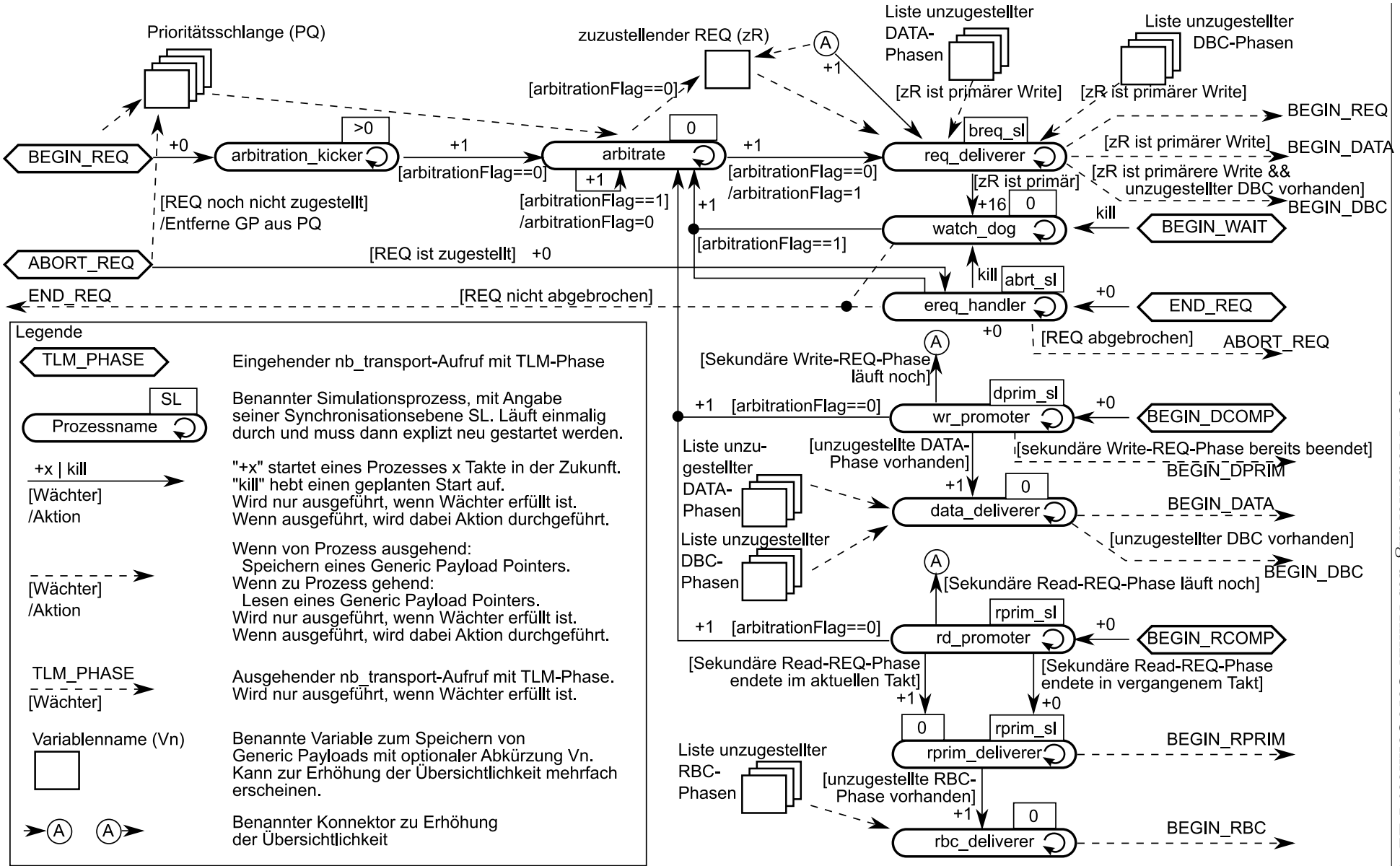


Abbildung 6.6.: Arbeitsweise der Arbitrierung im TLM-Modell des PLB v4.6

Danach wird das GP in einer Prioritätsschlange entsprechend der mit Hilfe der `priority` GP-Erweiterung übermittelten Priorität einsortiert, und der `arbitration_kicker` wird gestartet. Dieser stellt sicher, dass er auf einer Synchronisationsebene größer Null ausgeführt wird. Dadurch kann er sicher sein, dass der `arbitrate`-Prozess bereits ausgeführt wurde, denn der Kicker benötigt den aktuellen Wert des `arbitrationFlag`[11], da er den `arbitrate` Prozess nur dann startet, wenn der simulierte Addressbus frei ist.

Der `arbitrate`-Prozess wählt den höchst-prioren Request aus der Prioritätsschlange aus und legt ihn als primären oder sekundären Read- oder Write-Request fest. Ist dies nicht möglich, da zurzeit sowohl ein primärere als auch ein sekundärer Request laufen, passiert nichts weiter. Danach bestimmt er das ausgewählte GP als den zuzustellenden Request und startet einen Takt später den `req_deliverer`-Prozess. Der `arbitrate`-Prozess verhindert nun mit Hilfe des `arbitrationFlag` das Ausführen weiterer Arbitrierungssvorgänge und ist damit beendet. Der `req_deliverer` stellt den Request an den Slave zu, tut dies aber erst auf Synchronisationsebene `breq_sl` (siehe dazu Abschnitt 6.3.2). Handelt es sich dabei um einen primären Write-Request werden zusätzlich noch die entsprechende Daten-Phase[12] und gegebenenfalls eine ausstehende DBC-Phase zugestellt. Danach plant der `req_deliverer` einen Time-Out (Start des `watch_dog`-Prozesses), falls der zugestellte Request ein primärere Request war. Danach geschieht aus Sicht des PLB-Modells nichts weiter.

Startet der `watch_dog`-Prozess, sendet er ein `END_REQ` für den zuzustellenden Request zum entsprechenden Master, wobei der `end_type` auf Time-Out gesetzt wird. Anschliessend startet er einen Takt später den `arbitration`-Prozess. Dieser wird dann das `arbitrationFlag` zurücksetzen und sich selbst einen weiteren Takt später starten, um eventuell noch in der Prioritätsschlange enthaltene Requests zu arbitrieren. Die Transaktion, die mit dem GP modelliert wurde, das der zuzustellende Request ist, ist damit beendet.

Wird ein `END_REQ` vom Slave oder eine `ABORT_REQ` vom Master für den zugestellten Request empfangen, wird der Prozess `ereq_handler` auf Synchronisationsebene `abrt_sl` gestartet (siehe dazu Abschnitt 6.3.2). Dieser Prozess überprüft, ob ein Abbruch vorliegt. Ist dem so wird nur der Abbruch übermittelt, ein gleichzeitiger `END_REQ` wird verworfen. Ansonsten wird `END_REQ` zum entsprechenden Master gesendet und im Falle eines `rearbitrate-END_REQ` wird die Transaktion als beendet angesehen, und die für die einmalige Arbitrierungs-Rückstellung des Masters notwendigen Aktionen werden durchgeführt.

Wird für eine Transaktion eine DCOMP- oder RCOMP-Phase empfangen, starten die entsprechenden `promoter`-Prozesse. Diese befördern einen eventuell vorhandenen sekundären Request zum primären. Dies geschieht entweder mit Hilfe der jeweiligen PRIM-Phase, falls der Request bereits beendet ist, oder indem der Request erneut, diesmal aber als primärer Request zugestellt wird. Die `promoter`-Prozesse starten auch den `arbitrate`-Prozess erneut, falls der Adressbus frei ist, da dadurch auch wieder Platz für einen sekundären Request

[11] Das `arbitrationFlag` zeigt an, ob zurzeit eine Request-Phase zugestellt, aber noch nicht beendet wurde.

[12] Bei Einhalten des Protokolls ist diese zu diesem Zeitpunkt in der Liste unzugestellter Daten-Phasen gespeichert.

ist. Diese Prozesse werden auf speziellen Synchronsiationsebenen `rprim_sl` bzw. `dprim_sl` ausgeführt (siehe dazu Abschnitt 6.3.2).

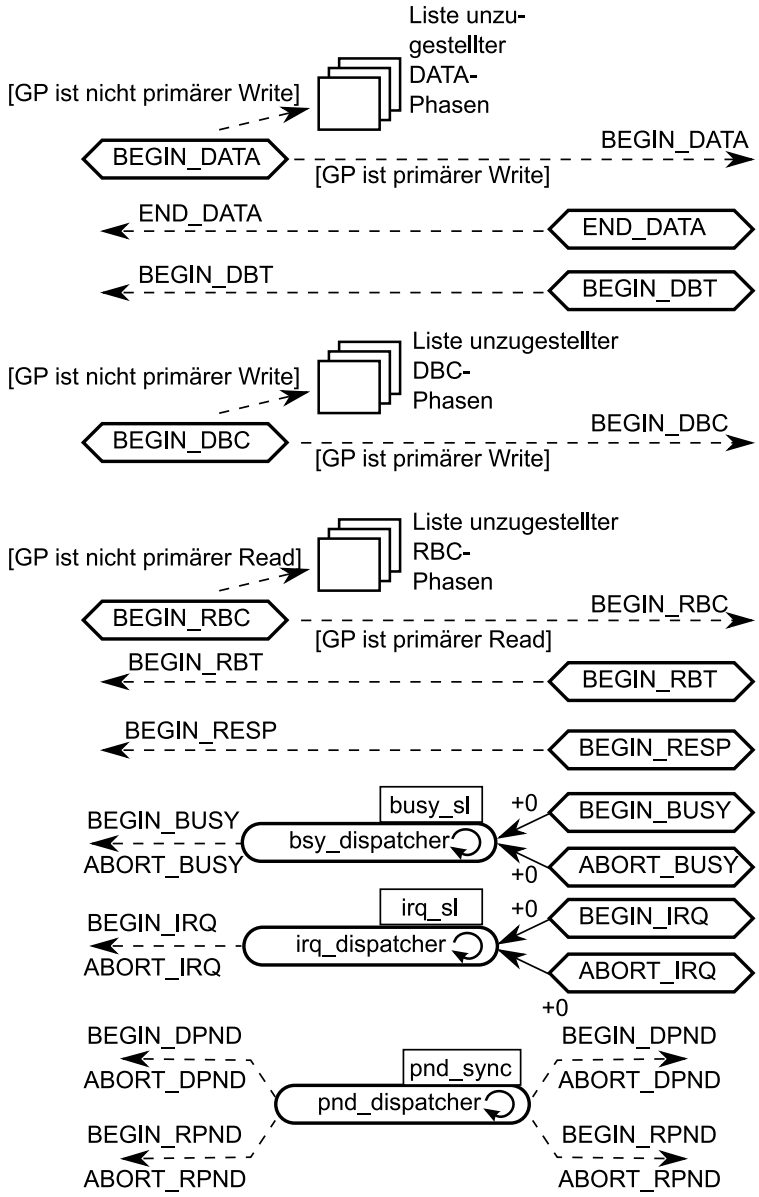


Abbildung 6.7.: Arbeitsweise der Datenpfade im TLM-Modell des PLB v4.6

Abbildung 6.7 zeigt die weiteren Datenpfade im TLM-Modell des PLB. Man erkennt, dass zur Behandlung der Schreib- und Lesedatenpfade keine Prozesse notwendig sind. Die eingehenden `nb_transport`-Aufrufe werden grundsätzlich direkt weitergeleitet. Zur Behandlung der BUSY- und IRQ-Phasen existiert je ein Prozess. Diese Prozesse werden gestartet, sobald irgendein Slave eine Änderung im Zustand der BUSY-Phase meldet. Dabei wird mit der entsprechenden Synchronsiationsebene sichergestellt, dass bei Ausführung des Prozesses alle Slaves (wenn überhaupt) solche Änderungen signalisiert haben. Die `dispatcher`-Prozesse senden dann die entsprechenden `BEGIN_`- oder `ABORT_`-Phasen mit einem speziellen GP, dessen `bsy_set`- bzw. `irq_set`-GP-Erweiterung entsprechend aller empfangener Daten gesetzt ist, zu allen angeschlossenen Mastern.

Der pnd_dispatcher-Prozess verteilt die aktuelle Information über die anhängigen Requests zu allen Mastern und Slaves. Der Prozess wird von vier Prozessen der Arbitrierung (Abbildung 6.6) gestartet: Direkt wenn der arbitrate-Prozess das arbitrationFlag zurücksetzt, wenn also ein Request vorbei ist; direkt wenn der arbitration_kicker ausgeführt wird, wenn also irgendein Master einen neuen Request gesetzt hat; einen Takt nach wr_promoter, wenn also gegebenenfalls kein sekundärer Write mehr existiert, und einen Takt nach rd_promoter, wenn also gegebenenfalls kein sekundärer Write mehr existiert. Ausgeführt wird der Prozess einmalig auf Synchronisationsebene pnd_sync (siehe Abschnitt 6.3.2).

Die Implementierung ist einfacher und übersichtlicher als das VHDL-Pendant. Dies liegt vor allem daran, dass das VHDL-Modell für die Logiksynthese vorgesehen ist. Das TLM-Modell kann zum Beispiel mit Hilfe der TID oder Synchronisationsebenen die Prozessausführung ordnen und so viele schwierig zu handhabende Situationen, besonders wenn verschiedene Phasen gleichzeitig starten oder enden, von vornherein vereinfachen. Darüber hinaus ist durch die Verwendung des GP als Container nicht immer eine Vielzahl von Informationen zu übergeben, sondern nur der Container. Daraus können dann alle notwendigen Informationen bei Bedarf extrahiert werden. Multiplexing von Transaktionen wird aufgrund der Natur von TLM-2.0 (Weiterleiten von GP-Referenzen) vermieden. Broadcasts von GPs sind mit Hilfe der Definition der Busphasen weitestgehend unterdrückt worden (lediglich die BUSY-, IRQ- und PND-Phasen müssen an alle Module gesendet werden). Die Implementierung des Verhaltens des Modells, also die Prozesse und IMCs, exklusive der VCD-Unterstützung, umfasst nur knapp 1000 Zeilen Code.

6.3.2. Partielle Prozess-Ausführungsordnung

Im vorangegangenen Abschnitt wurden diverse Synchronsiationsebenen erwähnt, welche hier nun erläutert werden sollen. Das ausführliche Code-Beispiel dazu kann Anhang K entnommen werden.

Die Synchronisationsebene breq_sl für den req_deliverer-Prozess (siehe Abbildung 6.6) ist eine Ebene höher als die spätest mögliche ABORT_REQ-, BEGIN_DATA- oder BEGIN_DBC-Phase. Dies ist notwendig, damit ein Abbruch im gleichen Takt, in dem er eigentlich zugestellt werden würde, vor der Zustellung passiert und diese somit verhindert. Darüber hinaus muss der Prozess auch die DATA- und eventuell auch eine DBC-Phase für einen primären Write-Request übermitteln. Dazu muss sichergestellt sein, dass diese auch bereits empfangen wurden.

Die Synchronisationsebene abrt_sl für den ereq_handler-Prozess (siehe Abbildung 6.6) ist eine Ebene höher als die spätest mögliche ABORT_REQ- oder END_REQ-Phase. Dies ist notwendig, damit der Prozess sicher sein kann, dass ein akzeptierter Request nicht noch später abgebrochen wird oder andersherum.

Die Synchronisationsebenen rprim_sl bzw. dprim_sl liegen eine Ebene über dem spätesten END_REQ und entsprechendem COMP, da das Verhalten der Promoter-Prozesse davon

abhängt, ob die sekundären Requests schon akzeptiert wurden (eventuell im gleichen Takt wie das COMP). Deshalb muss sicher gestellt sein, dass `END_REQ` bzw. das entsprechende COMP schon empfangen wurden. Für `dprim_sl` wird auch noch die höchste Synchronisationsebene von `BEGIN_DATA` herangezogen, da der `wr_promoter`-Prozess die Existenz einer bereits empfangenen Datenphase prüft, diese also schon (wenn überhaupt) begonnen haben muss.

Die Synchronisationsebenen `bsy_sl` bzw. `irq_sl` liegen eine Ebene über dem spätesten `[BEGIN|ABORT]_[BUSY|IRQ]`, da die endgültigen Werte für `bsy_set` bzw. `irq_set` in einem Takt die Bit-weise Veroderung aller `bsy_set`s bzw. `irq_set`s der Slaves sind und daher erst berechnet werden können, wenn alle feststehen.

Die Synchronisationsebenen `pnd_sl` liegt eine Ebene über dem spätesten `BEGIN_REQ` da erst, wenn alle Requests in einem Takt empfangen wurden, die Werte für die höchste Priorität feststehen.

6.3.3. Vorteile gegenüber der RTL-Implementierung

Hauptziel des TLM-Modells ist eine deutlich höhere Simulations-Performance als die der RTL-Simulation bei gleichen Aussagen zur Kommunikations-Performance. Welche Aspekte des Modells tragen dazu wesentlich bei?

1. Da es in der *J-R*-Simulation nur relevante Informationen gibt, gibt es keine unnötigen Prozessaktivierungen. Daher arbeitet das Modell rein reaktiv und verwendet Event-Chains, um seine internen Prozesse bei Aktivität an den Takt zu „klinken“. In Takten, in denen nicht kommuniziert wird, konsumiert das Modell keine Rechenleistung. Im Gegensatz dazu hat das RTL-Modell 29 getaktete Prozesse, die in jedem Takt einmal ausgeführt werden, unabhängig davon, ob kommuniziert wird.

2. Im Rahmen der taktgenauen busphasenbasierten *J-R*-Simulation mit TLM-2.0 werden alle Interface-Signale auf das Paar aus GP und TLM-Phase abgebildet (siehe Abschnitt 4.3). Ein signifikantes Ereignis (Auftreten eines Phasenrandes) reduziert sich auf ein Event im Simulator, während in RTL jedes gesetzte Signal zur Auslösung von Ereignissen führt.

3. Aufgrund der Eigenschaften von Busphasen sind nur deren Starts, Enden und Abbrüche interessant. Takte ohne solche Ereignisse können übersprungen werden.

4. Die Definition der REQ_{SL}-Phase (Abschnitt 6.2) erlaubt eine Zentralisierung der Addressdekodierung. Broadcasts von Requests sind nicht notwendig, und eine Addressdekodierung in jedem Slave (von denen in der Regel nur eine erfolgreich ist) entfällt.

5. Simulationsprozesse sind nur an der Arbitrierung beteiligt. Während des Datenaustauschs werden Funktionsaufrufe direkt weitergeleitet. Im RTL-Modell durchlaufen die Daten kombinatorische Multiplexer, es werden also Simulationsprozesse genutzt, um das Ergebnis zu errechnen.

6. Es werden nur Referenzen und Zeiger bewegt, während in RTL jeder Wert von Signal zu Signal kopiert wird.

7. Das Modell greift nur einmalig beim Request auf Elemente des GP zu. In diesem Fall sind alle Werte aufgrund ihrer Änderungsintervalle vertrauenswürdig. Es werden nur die Adresse, die Priorität und das Command abgefragt. Die übrigen GP-Erweiterungen und GP-Grundelemente sind nur für den Slave von Bedeutung. Im RTL-Modell müssen auch all diese für den Bus unwichtigen Werte durch den Bus und diverse Multiplexer kopiert werden.

8. Da `BEGIN_RESP`-Phasen direkt weitergeleitet werden, muss auch kein Zugriff auf das p2p-variante Feld `m_response_status` erfolgen.

9. Im Mittel sind nur vier bis sechs Prozesse gleichzeitig aktiv. Beim RTL-Modell sind es mindestens 29 getaktete Prozesse zuzüglich diverser kombinatorischer Prozesse bei Kommunikationsaktivität.

10. Die partielle Prozess-Ausführungsordnung garantiert die Vermeidung von Mehrfachausführungen von kombinatorischen Berechnungen.

6.3.4. Simulationsresultate

Die Simulation des in Abschnitt 6.3.1 beschriebenen PLB-Modells muss zwei Resultate liefern: Erstens muss die Simulation bezüglich der definierten Busphasenränder mit der RTL-Simulation taktgenau übereinstimmen, und zweitens muss sie dies bei weitem effizienter als die RTL-Simulation tun. Nur wenn beide Anforderungen erfüllt sind, kann von einer erfolgreichen Modellierung gesprochen werden.

Sicherstellung der busphasenbasierten J-R-Taktgenauigkeit

Um die Übereinstimmung bezüglich der Berechnung der Busphasenränder (Starts, -enden und -abbrüche) zu belegen, wurden die bereits in der Einleitung von Abschnitt 6.3 erwähnten eingeschränkt zufälligen Teststimuli verwendet. Diese wurden während der Entwicklung manuell erstellt und manuell verglichen. Dabei wurden meist kritische Corner-Cases behandelt. Später wurde dieser Prozess weiter automatisiert, um einerseits ein breites Spektrum von Testfällen zwischen den Corner-Cases zu untersuchen und andererseits, um automatisierte Benchmarks mit vielen verschiedenen Konstellationen zu erzeugen. Ein Beispiel für einen solchen Test zur Übereinstimmung bezüglich der Busphasen wird in Anhang L gezeigt.

Es wurden circa 100 manuelle Testsfälle und mehrere Tausend eingeschränkt zufällige Tests durchgeführt, von denen alle erfolgreich waren.

Simulations-Performance

Nachdem die korrekte Funktion des Modells durch eine Vielzahl von den oben erläuterten Tests als ausreichend belegt angesehen wurde, wurden diverse Untersuchungen zur Simula-

tions-Performance des Modells gemacht und mit einer funktional identischen RTL-Simulation verglichen. Dabei wurde auch sichergestellt, dass die berechneten Phasenränder in beiden Simulationen übereinstimmen.

Bisher wurde stets davon gesprochen, dass die TLM-Simulation signifikant schneller sein soll als die RTL-Simulation. Jedoch stellt sich die Frage, was signifikant schneller eigentlich heißt. Aus diesem Grund formuliere ich als konkretes Ziel, dass die TLM-Simulation mindestens eine Größenordnung schneller sein, also in höchstens einem Zehntel der Laufzeit der RTL-Simulation zu den gleichen Aussagen bezüglich der Busphasenstarts, -enden und abbrüchen kommen muss. Dabei soll diese Obergrenze (ein Zehntel der RTL-Laufzeit) mit genügendem Abstand unterschritten werden, um auch im Fall anderer Randbedingungen[13] die Beschleunigung um eine Größenordnung mit hoher Wahrscheinlichkeit zu gewährleisten.

In den im Folgenden erläuterten Experimenten werden vier verschiedene Simulationen miteinander verglichen: Zwei der vier Simulationen wurden mit dem VHDL-Modell des PLB v4.6 [XILI10] und entsprechenden RTL-Master- und Slave-Modulen durchgeführt. Dabei wurden bei beiden Simulationen die maximal mögliche Optimierung seitens des Simulators [MtGr08] zugelassen, jedoch wurde einmal die Sichtbarkeit modulinterner Signale erhalten (Option `+acc`), während im anderen Fall nur noch externe Signale sichtbar blieben.

In den beiden anderen Simulationen wurde das PLB-TLM-Modell aus Abschnitt 6.3.1 mit entsprechenden TLM-Master- und Slave-Modulen verwendet. Einmal wurde die Simulation mit einem gemäß Abschnitt 5 modifizierten SystemC-Kernel betrieben, sodass Event-Chains, Synchronsiationsebenen und die Taktung mit Kernel-Modifikation verwendet werden konnten. Im anderen Fall wurde ein Kernel eingesetzt der lediglich Event-Chains und Synchronsiationsebenen unterstützt[14]. In diesem Fall wurde die Taktung nach Grellier verwendet.

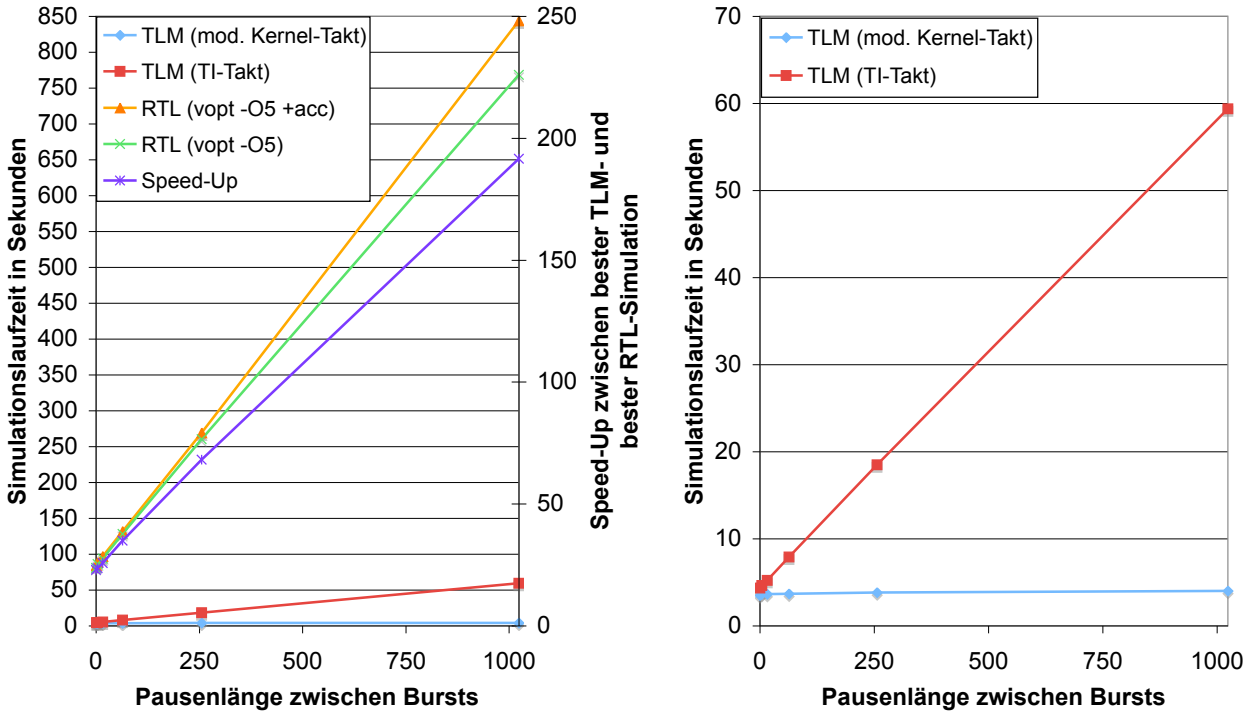
Die RTL- beziehunsgweise TLM-Master- und Slave-Module sind, wie in der Einleitung von Abschnitt 6.3 beschrieben, konfigurierbar. Um den Einfluss der Modellierung der Master und Slaves auf die Messungen zu minimieren, wurden deren Modelle für die RTL-Simulation in einer nicht Logiksynthese-fähigen Art und Weise implementiert. So verwenden sie explizite Events oder auch Warte-Anweisungen innerhalb der Prozesse. Dies erlaubt eine effizientere Modellierung der Module, obwohl es sich dann streng genommen nicht um RTL-Modelle der Master und Slaves handelt.

Im ersten Experiment wurde der Einfluss der Länge der Pause, die ein Master zwischen zwei Kommunikationen macht, auf die Simulations-Performance untersucht. Die Ergebnisse sind in Abbildung 6.8 auf der nächsten Seite dargestellt. Man erkennt, dass die TLM-Simulation mit Kernel-Modifikation die beste Performance zeigt, wobei aber auch deutlich wird, dass die taktgenaue *J-R*-Simulation ohne Kernel-Modifikation bereits einen großen Performance-Vorteil gegenüber RTL hat. Die allgemeinen Gründe dafür sind in Abschnitt 6.3.3 gelistet, im weiteren Verlauf dieses Abschnittes werden nur noch die darüber hinaus-

[13] Z.B.: Ein anderer Rechner, auf dem die Simulation ausgeführt wird, eine andere SystemC-Implementierung oder ein anderer RTL-Simulator.

[14] Diese Modifikationen erleichtern die Modellierung und vermeiden Simulationszeitartefakte, haben aber keinen wesentlichen Einfluss auf die Performance.

gehenden Besonderheiten der Ergebnisse erläutert. Zum Beispiel zeigt sich in Abbildung 6.8b erneut der große Vorteil des effizienten Überspringens ungenutzter Takte. Bei der Taktung mit Hilfe der Kernel-Modifikation hat die Anzahl der übersprungenen Takte keinen Einfluss auf die Simulationsdauer. Der Anwendungsfall mag konstruiert wirken, hat aber durchaus Relevanz: Im Rahmen einer Trace-basierten oder stochastischen Simulation[15] eines Einzelprozessor-Systems, wie es zum Beispiel in simplen eingebetteten Systemen häufig der Fall ist, kommt es bei der Simulation von Software mit hoher Datenlokalität nur zu sporadischer Kommunikationsaktivität (bei simulierten Cache-Misses). In solchen Fällen kommt es leicht zu mehreren tausend ungenutzten Takten zwischen Kommunikationen.



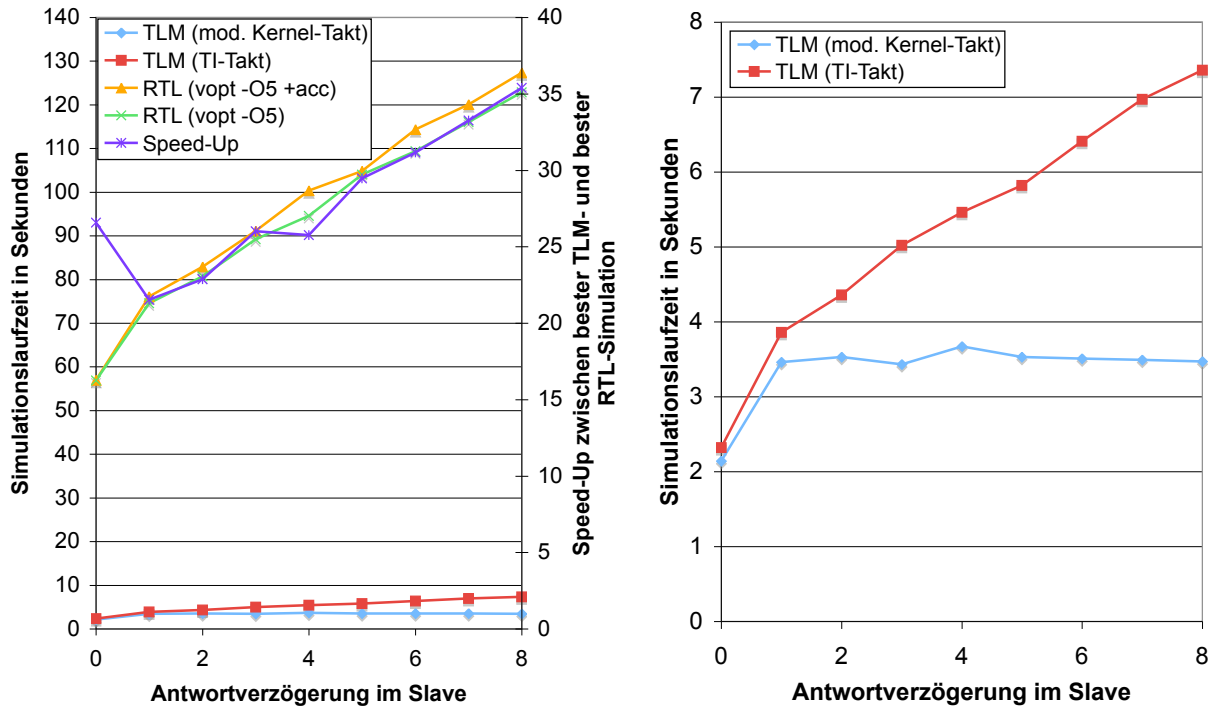
(a) inklusive RTL und Speed-Up

(b) exklusive RTL und Speed-Up

Abbildung 6.8.: Simulationslaufzeiten für 250.000 8-Wort-Write-Bursts von einem Master zu einem Slave, mit zwei Takten Antwortverzögerung im Slave in Abhängigkeit von der Pause zwischen den Bursts

In einem weiteren Experiment wurde der Einfluss der Antwortverzögerung im Slave auf die Simulations-Performance vermessen. Die Ergebnisse zeigt Abbildung 6.9 auf der nächsten Seite. Nach den Erfahrungen aus dem ersten Experiment ist auch hier erwartungsgemäß die TLM-Simulation deutlich effizienter. Interessant ist der Einbruch im Speed-Up, wenn die Antwortverzögerung von Null auf Eins steigt. Die Erklärung dafür liefert Abbildung 6.9b. Die Simulationszeit steigt sprunghaft, wenn der Slave seine Antwortverzögerung von Null

[15] Die Master am Bus werden dabei durch Modelle ersetzt, die vorher in einem realen System aufgezeichnete Kommunikationsabläufe „abspielen“ (Trace-basierten Simulation) oder die mit Hilfe von konfigurierbaren, stochastischen Modell das Verhalten der Master wiederspiegeln (stochastische Simulation). Dies ist ein realistisches Vorgehen, auch im Rahmen der taktgenauen Analyse [Aldi06].



(a) inklusive RTL und Speed-Up

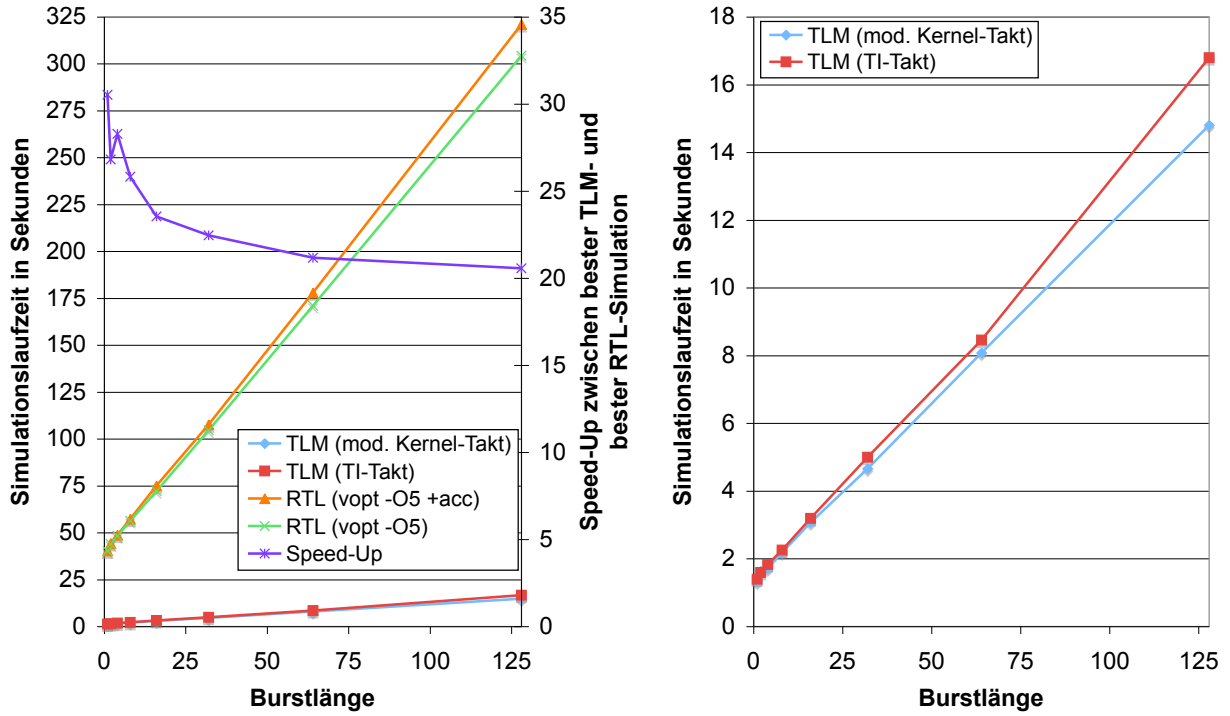
(b) exklusive RTL und Speed-Up

Abbildung 6.9.: Simulationslaufzeiten für 250.000 8-Wort-Write-Bursts von einem Master zu einem Slave, ohne Pause zwischen den Bursts in Abhängigkeit von der Antwortverzögerung im Slave

auf Eins ändert, danach steigt sie im Falle der Kernel-Modifikation nicht weiter. Der Grund dafür ist, dass bei keiner Verzögerung der Slave `TLM_UPDATED` verwendet und folglich keinen internen Prozess starten muss, um aktiv `nb_transport_bw` zu verwenden. Will der Slave seine Antwort jedoch verzögern, so startet er mit Hilfe einer Event-Chain einen internen Prozess[16]. Da ungenutzte Takte effizient übersprungen werden, hat die eigentliche Länge der Verzögerung keinen weiteren Einfluss auf die Simulations-Performance.

Im nächsten Experiment wurde der Einfluss der Länge einer Transaktion auf die Simulations-Performance untersucht. In Abbildung 6.10 auf der nächsten Seite sind die entsprechenden Ergebnisse gezeigt. In diesem Fall skalieren alle Laufzeiten linear, da eine steigende Burstlänge eine steigende Anzahl von Phasen pro Transaktion bedeutet. Der Speed-Up konvergiert gegen 20. Die höheren Speed-Ups bei kurzen Bursts begründen sich hierbei dadurch, dass die Arbitrierung im TLM-Modell deutlich effizienter simuliert werden kann als in logiksynthetisierbaren RTL-Strukturen. Ab einer gewissen Länge dominiert dann aber, dass in jedem Takt eine Datenphase auftritt. Der Ausreißer bei Burstlänge Vier resultiert aus den sehr kurzen Laufzeiten der TLM-Simulation. Eine Schwankung innerhalb der Messungenauigkeit, die im Verlauf in Abbildung 6.10b gar nicht auffällt, führt in der Division zur Bestimmung des Speed-Ups zu einer solchen Schwankung.

[16]Gemäß der Empfehlung aus Abschnitt 4.4 wird das `time`-Argument nicht verwendet.



(a) inklusive RTL und Speed-Up

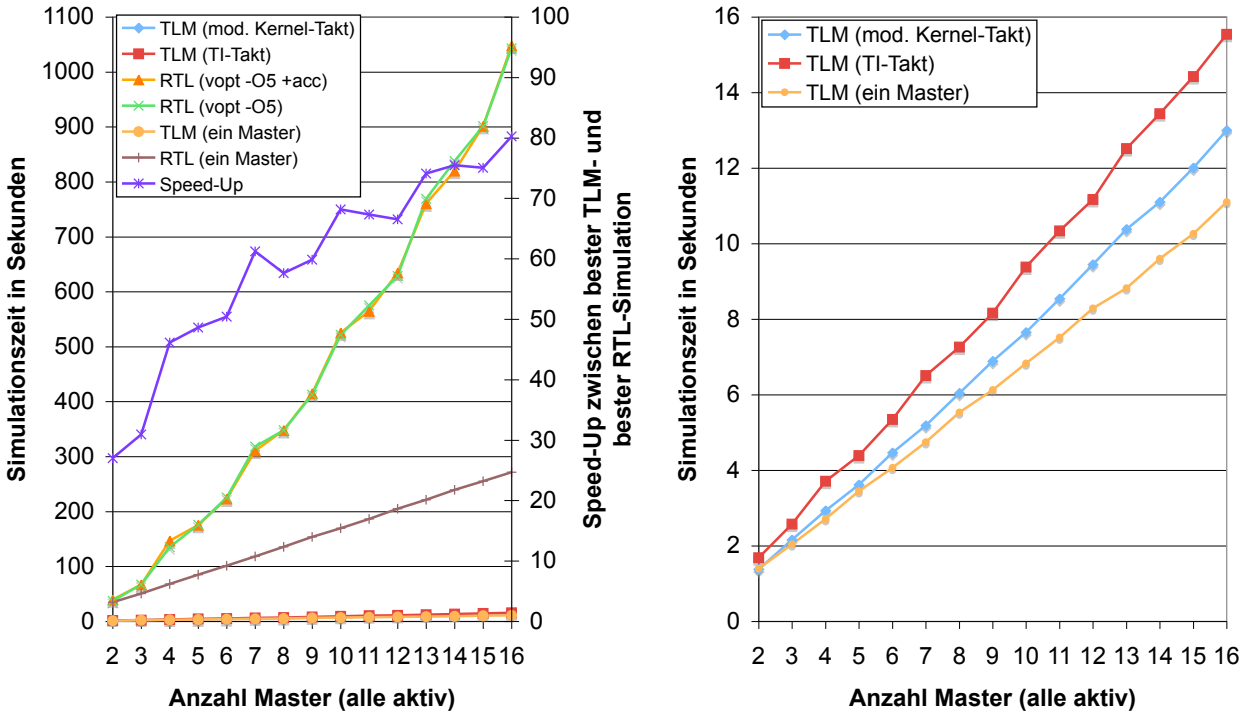
(b) exklusive RTL und Speed-Up

Abbildung 6.10.: Simulationslaufzeiten für 250.000 Write-Bursts von einem Master zu einem Slave, ohne Pause zwischen den Bursts, ohne Antwortverzögerung im Slave in Abhängigkeit von der Burstlänge

In dem Experiment, dessen Ergebnis in Abbildung 6.11 auf der nächsten Seite dargestellt ist, wurde die Anzahl gleichzeitig aktiver Master erhöht. In allen Fällen skaliert die Laufzeit grob linear. Dies begründet sich in erster Linie dadurch, dass bei fester Anzahl von Transaktionen pro Master und steigender Anzahl von Mastern die Anzahl der zu simulierenden Transaktionen steigt. Die Simulations-Performance der Taktung mit modifiziertem Kernel ist aufgrund des effizienteren Überspringens freier Takte messbar besser als die Taktung ohne Kernel-Modifikation.

Zusätzlich zu den vier bereits bekannten Messreihen enthält Abbildung 6.11 noch zwei weitere Messreihen: „RTL (ein Master)“ und „TLM (ein Master)“ entsprechen den Simulationen für „RTL (vopt -O5)“ beziehungsweise „TLM (mod. Kernel-Takt)“, jedoch steigt diesmal nicht die Anzahl der Master. Es gibt nur einen Master, dessen Anzahl durchzuführender Transaktionen steigt. Beim Messpunkt 6, zum Beispiel, finden $6 * 45.000 = 270.000$ Transaktionen statt.

Man erkennt, dass bei der TLM-Simulation die Laufzeit des einzelnen Master nur geringfügig schwächer skaliert als die der steigenden Masteranzahl. Grund dafür ist, dass prinzipiell stets nur ein Master aktiv ist (der aktuell primäre Write), während alle anderen Master warten. Dieses Warten verursacht in der TLM-Simulation keine zusätzlichen Kosten, da die Master dank der Natur der *J-R*-Simulation reaktiv mit Event-Chains auf eine neue Aktivierung warten. Die etwas schwächere Skalierung entsteht, da im Ein-Master-System keine Ar-



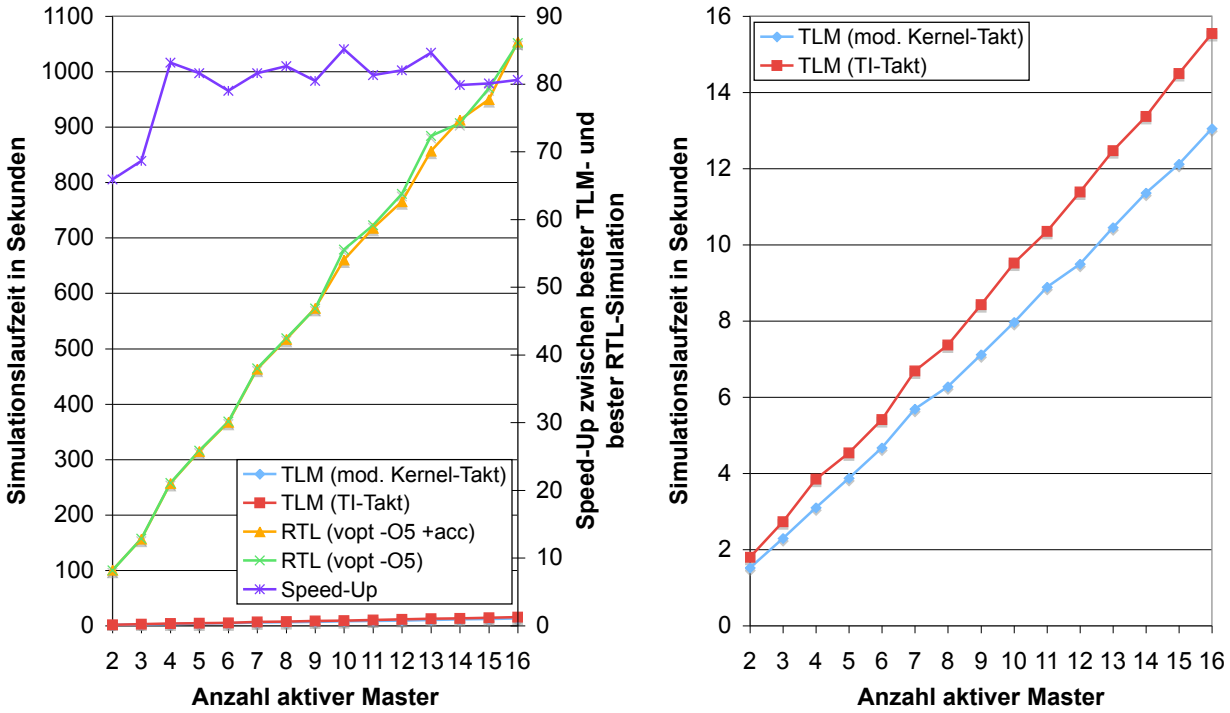
(a) inklusive RTL und Speed-Up

(b) exklusive RTL und Speed-Up

Abbildung 6.11.: Simulationslaufzeiten für 45.000 8-Wort-Write-Bursts pro Master zu einem Slave, mit 16 Takten Pause zwischen Bursts und 2 Takten Antwortverzögerung im Slave in Abhängigkeit von der Anzahl Master; alle Master aktiv

bitrierung simuliert werden muss und weil die DPND-, RPND-, BUSY- und IRQ-Broadcasts weniger Kosten verursachen. In der RTL-Simulation wiegt die Anzahl der Master deutlich schwerer. Grund dafür ist, dass mit steigender Anzahl von Mastern die Anzahl von getakteten Prozessen steigt (ein aktiver Master benötigt einen Prozess, der die eingehenden Signale taktsynchron abfragt), und innerhalb des PLB steigt die Breite der Multiplexer rapide an. Das TLM-Modell skaliert also größenordnungsmäßig nur mit der Busauslastung und kaum mit der Anzahl der Master. Dies ist ein wichtiger Vorteil bei der Simulation von Multi-Master-Systemen (z.B. Multi-Prozessorsysteme mit DMA-Controller). Die starken Schwankungen im Speed-Up ergeben sich aus dem nicht konstanten Anstieg der Simulationslaufzeit der RTL-Simulation. Als Grund hierfür vermute ich, dass Multiplexer und Oder-Verknüpfungen verschiedener Breiten unterschiedlich gut optimiert werden können und ähnliche Effekte.

Abbildung 6.12 auf der nächsten Seite zeigt die Ergebnisse eines ähnlichen Experiments, jedoch ist diesmal die Anzahl der Master konstant 16 und die Anzahl der davon aktiven Master wird erhöht. Man erkennt, dass die Werte für Messpunkt 16 in Abbildung 6.12 denen aus Abbildung 6.11 am gleichen Messpunkt entsprechen, weil in diesem Fall je 16 aktive Master existieren; die beiden Messungen somit identisch sind. Interessanter sind die vorhergehenden Messpunkte. In der TLM-Simulation ist kaum ein Unterschied messbar. Die geringfügig längere Laufzeit (ca. 3 Prozent) bei weniger aktiven Mastern ergibt sich dadurch, dass in diesem Experiment die Kosten der DPND-, RPND-, BUSY- und IRQ-Broadcasts höher lie-



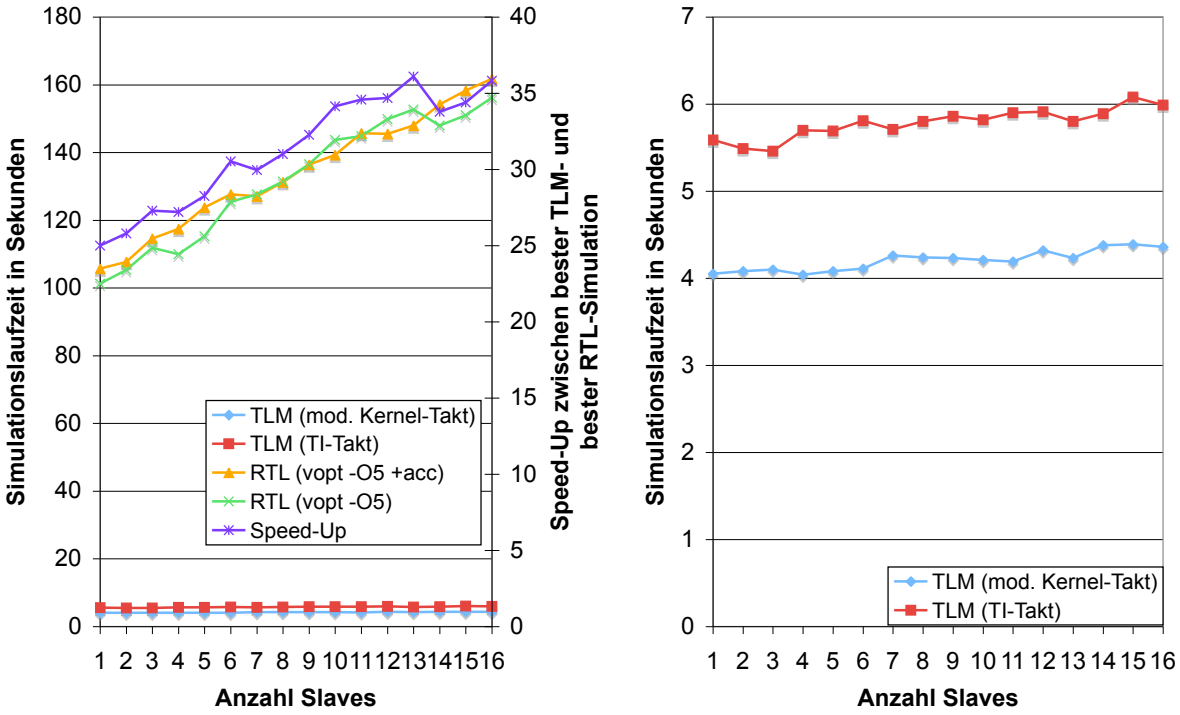
(a) inklusive RTL und Speed-Up

(b) exklusive RTL und Speed-Up

Abbildung 6.12.: Simulationslaufzeiten für 45.000 8-Wort-Write-Bursts pro aktivem Master zu einem Slave, mit 16 Takten Pause zwischen den Bursts, mit 2 Takten Antwortverzögerung im Slave in Abhängigkeit von der Anzahl aktiver Master, wobei immer 16 Master im System sind

gen als bei dem Experiment, bei dem es keine inaktiven Master gab. Anders sieht es bei der RTL-Simulation aus. Die Laufzeit bei wenig aktiven Mastern ist im Vergleich zum vorangegangenen Experiment stark gestiegen (z.B. bei Messpunkt 3 von 56 Sekunden um über 200 Prozent auf 166 Sekunden). Die inaktiven Master haben keine aktiven Prozesse, sodass dieser Anstieg einzig aus dem PLB-Modell (zum Beispiel gestiegene Multiplexerbreite) resultiert. Diese Messung stützt die Aussage der vorangegangenen Messung, dass im taktgenauen busphasenbasierten *J-R*-Modell des PLB die Busauslastung die Performance dominiert, nicht die Anzahl beteiligter Module.

Das Ergebnis eines ähnlichen Experiments ist in Abbildung 6.13 auf der nächsten Seite dargestellt. In diesem Fall wurde die Anzahl der am Bus vorhandenen Slaves erhöht, jedoch wird von diesen immer nur ein Slave angesprochen. Dadurch ändert sich jetzt die Anzahl zu tätigender Transaktionen nicht, und die simulierte Dauer bleibt konstant. Damit sollen die steigenden Kosten der Broadcasts, vor allem die DPND-Broadcasts, genauer untersucht werden. In Abbildung 6.13b erkennt man, dass die Kurven zwar leicht aber kaum signifikant steigen (ein Anstieg der Laufzeit von der ersten zur letzten Messung um ca. 0,5 Prozent der ersten Messung pro zusätzlichem Slave). Broadcasts wie im PLB-Modell sind also sehr effizient. In der RTL-Simulation ist der Effekt gravierender. Auch hier sind die größeren



(a) inklusive RTL und Speed-Up

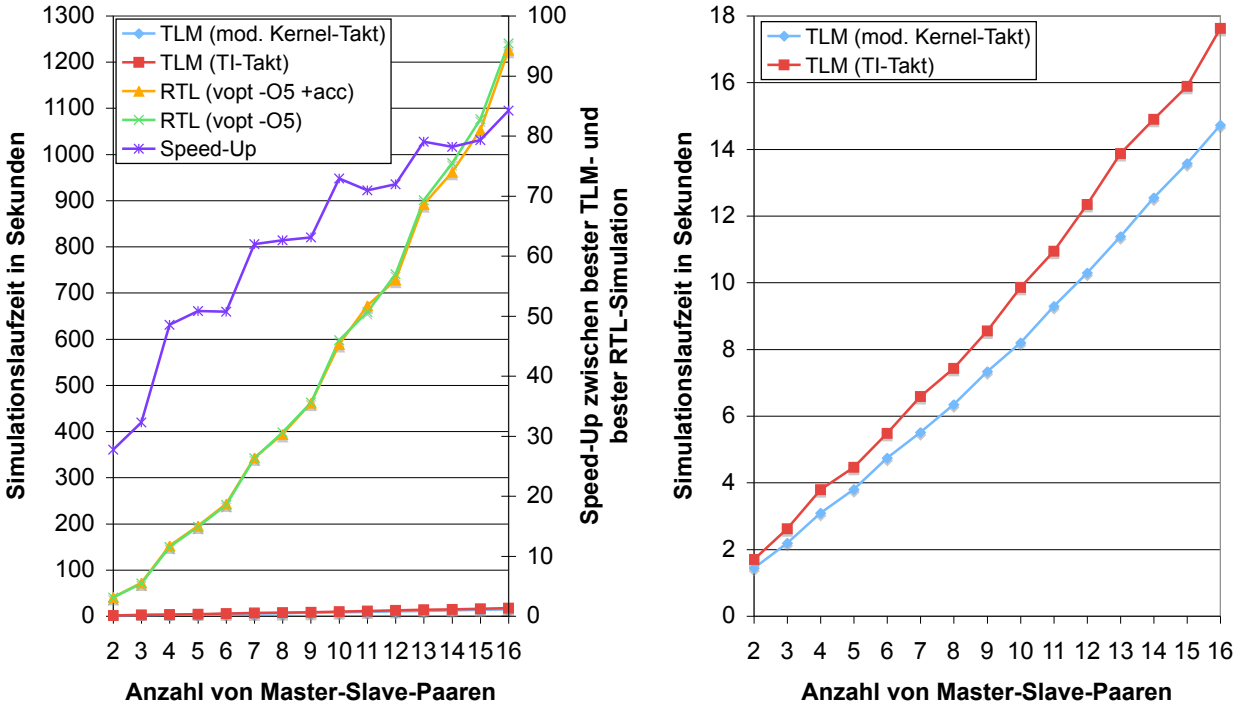
(b) exklusive RTL und Speed-Up

Abbildung 6.13.: Simulationslaufzeiten für 250.000 8-Wort-Write-Burstsvon einem Master zu einem Slave, mit 16 Takten Pause zwischen den Bursts, mit 2 Takten Antwortverzögerung im Slave in Abhängigkeit von der Anzahl von Slaves am Bus

Multiplexer und die höhere Anzahl der aktiven Prozesse im System aussschlaggebend. Jeder Slave besitzt einen Prozess, der bei Änderungen von `PAValid` oder `SAValid` eine Addressauswertung vornehmen muss. Im TLM-Modell ist dies nicht notwendig, da aufgrund der Art, wie die Busphasen definiert wurden, die entsprechenden `BEGIN_REQ`-Phasen nur zum addressierten Slave gehen. Die Addressdekodierung ist also im TLM-Modell zentralisiert worden und somit unabhängig von der Anzahl vorhandener Slaves (siehe auch Abschnitt 6.2, Hinweis zur Phase REQ_{SL} auf Seite 154).

Abschließend wurde eine Experiment durchgeführt, in dem die Anzahl der Master und Slaves im System gemeinsam erhöht wurde. Dabei versuchen immer je ein Master und ein Slave zu kommunizieren, es gibt also keine inaktiven Module im System. Es handelt sich um eine Kombination der Experimente aus Abbildungen 6.11 auf Seite 175 und 6.13 und das Ergebnis ist in Abbildung 6.14 auf der nächsten Seite dargestellt. Da die Anzahl der simulierten Transaktionen die gesamte Simulationslaufzeit dominiert, skalieren die Simulationen sehr ähnlich zu Abbildung 6.11, man kann aber auch erkennen, dass die Gesamtlaufzeiten (im Vergleich zu Abbildung 6.11) steigen. Da immer nur ein Master-Slave-Paar zu einem gegebenen Zeitpunkt aktiv ist, sind die anderen Slaves quasi inaktiv; die Effekte aus dem Experiment aus Abbildung 6.13 greifen. Das bedeutet, dass die Laufzeiten der TLM-Simulationen geringer

steigen, als die der RTL-Simulationen. Dies wird auch dadurch deutlich, dass der Speed-Up bei 16 Mastern und 16 Slaves höher ist als der bei 16 Mastern und einem Slave.



(a) inklusive RTL und Speed-Up

(b) exklusive RTL und Speed-Up

Abbildung 6.14.: Simulationslaufzeiten für 45.000 8-Wort-Write-Bursts von zwischen je einem Master und einem Slave, mit 16 Takten Pause zwischen den Bursts, mit 2 Takten Antwortverzögerung in den Slaves in Abhängigkeit von der Anzahl kommunizierender Master-Slave-Paare am Bus

Zusammenfassend wird in allen Experimenten deutlich, dass eine Beschleunigung um eine Größenordnung mit Hilfe der taktgenauen busphasenbasierten J-R-Simulation mit TLM-2.0 deutlich erreicht wird. Ein Speed-UP von 20 wurde nie unterschritten, im Gegenteil wurde in einer Messung ein Speed-Up jenseits von 100 (siehe Abbildung 6.8 auf Seite 172) erreicht. Trotzdem wird natürlich die Taktgenauigkeit nie verletzt, so wie es durch die Inputabstraktion J und das Relevanzkriterium R gefordert wird.

7. Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Vorschlag für einen auf TLM-2.0 basierenden Standard zur taktgenauen Transaction-Level-Modellierung von Kommunikationsstrukturen mit memory-mapped Bus-Interfaces präsentiert. Neben dem Standardisierungsvorschlag an sich wurden für den Anwender auch Regeln und Vorgehensanweisungen bestimmt, ein bisher nicht mit taktgenauer TLM modelliertes memory-mapped Bus-Interface auf den vorgeschlagenen Standard abzubilden. Diese Abbildungen und die im Standardvorschlag enthaltenen Implementierungs- und Verwendungsregeln zielen auf einfache Entwicklung, hohe Simulationsgeschwindigkeit und geringe Fehleranfälligkeit der taktgenauen TLM-Modelle ab. Da der Standardisierungsvorschlag auf TLM-2.0 basiert, finden einerseits Ingenieure mit Erfahrungen in TLM-2.0 einen einfachen Einstieg in die taktgenaue Modellierung, und andererseits können bereits existierende Werkzeuge wie zum Beispiel Transaktions-Logger weiterverwendet und Adapter zwischen taktgenauen und abstrakten Modellen vereinfacht werden.

7.1. Ergebnisse

Nach einer kurzen Einführung in das Forschungsfeld und den Aufbau dieses Dokuments gab Kapitel 2 einen Einblick in die Grundlagen dieser Arbeit.

Danach legte Kapitel 3 formal fest, was im Rahmen dieser Arbeit mit „taktgenau“ gemeint ist. Dazu wurden die Begriffe der Inputabstraktion J und der Relevanzauswahl R und eine darauf basierende taktgenaue J-R-Simulation definiert, sodass durch Kenntnis von J und R eine bestimmte Art der Taktgenauigkeit gegeben war. Da aber J und R praktisch schwer handhabbar waren, wurden in Kapitel 3 die wichtigsten Anwendungsgebiete taktgenauer TLM – namentlich Performance-Evaluation und Power-Analyse – analysiert und J und R dann anhand deutlich einfacherer Busphasen strukturiert. Diese Busphasen sind für verschiedene Teile des Bus-Interfaces definiert und beschreiben einen einfachen Datenaustausch von einem Sender zu einem Empfänger, zum Beispiel die Übergabe des Kommandos und der Adresse vom Master zum Slave oder die Übergabe der Lesedaten vom Slave zum Master. Eine daraus konstruierte J-R-Simulation wird dann die Starts, Enden oder Abbrüche dieser Busphasen taktgenau berechnen.

Der Vorteil der Busphasen liegt darin, dass sie im Rahmen der memory-mapped Bus-Interfaces ein recht natürliches Konzept sind. Ingenieure, die ein Busprotokoll auf RTL-Ebene beherrschen, sind in der Lage, dafür einen Satz von Busphasen zu definieren. Dieser formal erfasste Satz von Busphasen kann dann an einen anderen Ingenieur übergeben werden,

der daraus das TLM-Interface konstruiert. Dadurch ist es nicht notwendig, dass ein Ingenieur sowohl RTL- als auch TLM-Experte ist. Experten auf den jeweiligen Gebieten können eingesetzt werden, wobei aber die in Kapitel 1 beschriebene Gefahr von Mißverständnissen zwischen beiden dank der formalen, trotzdem aber einfachen Struktur der Busphasen minimiert wird.

In Kapitel 4 wurde zunächst gezeigt, dass die Verwendung von TLM-2.0 nicht Selbstzweck, sondern eine gut begründete Entscheidung war. Danach wurden Regeln und Vorgehensanweisungen zur Abbildung von Busphasen auf die Datenstrukturen von TLM-2.0 bestimmt. Der zentrale Schritt zur Abbildung eines bisher nicht taktgenau TLM-modellierten memory-mapped Bus-Interfacess bestand dann „nur“ noch in der Bestimmung der Busphasen, da anschließend die erwähnten Regeln verwendet werden können, um ein TLM-Interface zu konstruieren.

Darauf aufbauend untersuchte Kapitel 4 weiter, ob TLM-2.0 – speziell die Regeln, die von TLM-2.0 für die Verwendung des Non-blocking-Transport-Interfaces, des Generic Payloads und der TLM-Phase bestimmt werden – für die taktgenaue Modellierung in jeder Hinsicht geeignet ist. Dabei wurden Unzulänglichkeiten bezüglich der Bindungs-Checks, der Modifiabilities im Generic Payload und der Unterstützung des Nutzers bei der Implementierung und Verwendung von Generic Payload-Erweiterungen identifiziert und Lösungen dafür präsentiert. Diese Lösungen haben keinen Einfluss auf die abstraktere Modellierung mit TLM-2.0; es sind also Erweiterungen und keine Änderungen von TLM-2.0.

Während die Erweiterung der Bindungs-Checks zwingend notwendig war, handelt es sich bei den Erweiterungen bezüglich der Modfiabilities im GP und der Implementierung und Verwendung von GP-Erweiterungen um Regelungen, die in erster Linie dem Nutzer die taktgenaue Modellierung vereinfachen und eine robuste und effiziente Simulation gewährleisten. Ein Standardisierungsvorschlag ohne diese Regeln ist denkbar, würde aber den Einstieg in die taktgenaue Modellierung erschweren und zur Frustration der Entwickler und zu weniger performanten Modellen führen. Die Erfahrungen in Foren und Workingroups haben gezeigt, dass die Anwender gerade solche Regeln vermissen und dass die Option der Ausdehnung dieser Regeln auf die abstrakteren Modellierungsstile AT und LT angedacht werden kann.

Abschließend diskutierte Kapitel 4 Probleme, die ausschließlich bei der taktgenauen Modellierung auftraten. Dies sind die Taktung und die Behandlung kombinatorischer Berechnungen, und auch dafür wurden Erweiterungen von TLM-2.0 präsentiert. Das entsprechend erweiterte TLM-2.0 bildete dann im Zusammenhang mit der in Kapitel 3 definierten taktgenauen busphasenbasierten *J*-*R*-Simulation meinen Vorschlag zur Standardisierung von taktgenauer TLM.

Nachdem als Ergebnis von Kapitel 4 ein erweitertes TLM-2.0 zur taktgenauen busphasenbasierten *J*-*R*-Simulationen existierte, untersuchte Kapitel 5, inwieweit SystemC erweitert werden kann, um solche Simulationen einfacher oder effizienter als mit einem IEEE1666-konformen SystemC zu betreiben. Die entsprechenden Erweiterungen wurden exemplarisch für den OSCI-Referenz-Simulator umgesetzt und die Auswirkungen untersucht.

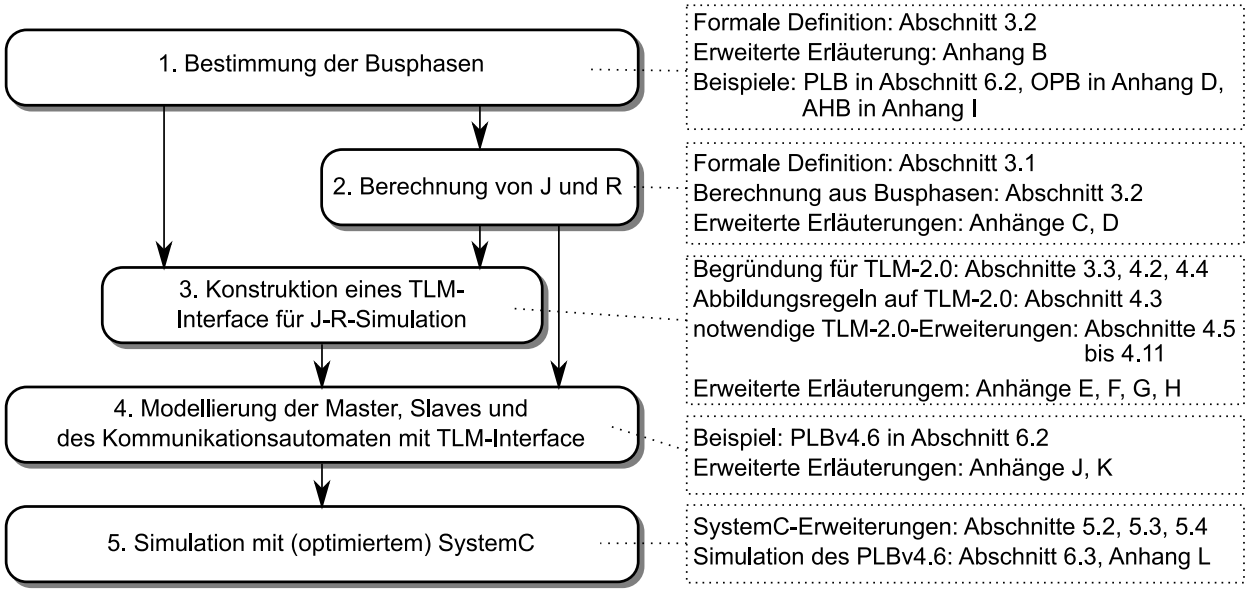


Abbildung 7.1.: Übersicht über die Beiträge dieser Arbeit (vergleiche Abbildung 3.15)

Dabei wurde deutlich, dass gerade für die speziell im Zusammenhang mit taktgenauer TLM auftretenden Probleme wie Taktung und kombinatorische Berechnungen noch Optimierungspotential besteht. Diese Optimierungen sind nicht immer auf einen Speed-Up der Simulation ausgelegt, sondern zielen auch darauf ab, die Modellierung der Probleme zu vereinfachen und die Gefahr entsprechender Modellierungsfehler zu reduzieren. Ein Slow-Down der Simulation ist dabei natürlich auf keinen Fall akzeptabel und konnte vermieden werden. Besonders herauszuheben ist die Optimierung der Taktunterstützung seitens des Simulationskernels. Es konnte eine Taktung implementiert werden, deren Simulations-Performance in keinem Test der Simulations-Performance irgendeiner anderen zurzeit bekannten Taktmodellierung unterlag. Im Gegenteil war sie in vielen Tests bei weitem effizienter als die jeweils beste bisher existierende Implementierung.

Die praktische Anwendung des vorgeschlagenen Standards wurde dann in Kapitel 6 illustriert. Zunächst wurden alle memory-mapped Bus-Interfaces gelistet, die zur Erkenntnisfindung oder Anwendbarkeitsuntersuchung im Rahmen dieser Arbeit herangezogen wurden. Anschließend wurde die taktgenaue busphasenbasierte Modellierung des IBM CoreConnect PLB zur Performance-Evaluation im Detail ausgearbeitet, und die Simulations-Performance des entstandenen Modells wurde mit einer entsprechenden RTL-Simulation verglichen.

Die Untersuchungen zeigen, dass das Erstellen eines taktgenauen TLM-Modells eines komplexen Busses mit Hilfe des vorgeschlagene Standards recht kompakt (ca. 1000 Zeilen Code) möglich ist. Dies soll aber nicht darüber hinwegtäuschen, dass das Erstellen selbst alles andere als trivial ist. Dies aber von der Erstellung eines taktgenauen Modells eines komplexen Busses zu fordern, ist auch illusorisch. Gerade deswegen ist die (halb-)automatische Unterstützung des Anwenders bei der Modellierung, also bei der Verwendung des vorgeschlagenen Standards, ein lohnendes Forschungsgebiet für die Zukunft (siehe auch Abschnitt 7.2).

Die Messungen zeigten klar, dass ein mit Hilfe des Standardisierungsvorschlages modellierter Bus die Simulations-Performance seines in klassischer RTL beschriebenen Pendants in jedem Fall, also selbst ohne die Optimierungen des Simulationskernels, um mindestens eine Größenordnung überbietet. Auf dem verwendeten Benchmark-Rechner wurde ein Speed-Up von 20 nie unterschritten. Mit der Optimierung konnten in einigen (stets realistischen) Fällen sogar Speed-Ups jenseits des Faktors 100 erreicht werden.

Die Beiträge der verschiedenen Kapitel dieser Arbeit sind noch einmal in Abbildung 7.1 zusammengefasst.

7.2. Ausblick

Die vorliegende Arbeit fokussiert auf memory-mapped Bus-Interfaces und deren taktgenaue TLM-Simulation. Dies wird in Abschnitt 2.3.1 begründet. Im Rahmen weiterführender Forschung ist es jedoch durchaus interessant, ob sich die vorgestellten Konzepte auch auf Kommunikationsstrukturen ohne memory-mapped Bus-Interfaces anwenden lassen, wie z.B. Feldbusse (Flexray™ oder CAN) oder serielle Hochgeschwindigkeits-Punkt-zu-Punkt-Verbindungen (SATA oder PCIe). Auch eine Auseinandersetzung mit asynchronen Protokollen wie dem VMEbus wäre von Interesse.

Die Fokussierung auf taktgenaue Modellierung unter Ausblendung der abstrakteren Modellierungsansätze begründet sich dadurch, dass für die Kopplung mit AT- und LT-Modellen in jedem Fall Adapter notwendig sind[1]. Unterschiede in den verwendeten GP-Elementen oder Erweiterungen können in diesen Adaptern ausgeglichen werden, da dies gemessen an der Gesamtlast meist mit nur sehr geringem Simulationsoverhead verbunden ist. Nichtsdestotrotz ist es lohnenswert, die im Rahmen dieser Arbeit definierten Regeln zur Bestimmung, Implementierung und zum Zugriff auf GP-Erweiterungen im Gesamtkontext über alle Modellierungsstile[2] hinweg neu zu bewerten und dadurch eventuell die Adaption zwischen den Modellierungsstilen noch weiter zu vereinfachen. Zu einem gewissen Teil ist dies bereits in den Untersuchungen zu $[\text{OCP-09}]_i$ und $[\text{CaGr10}]_i$ geschehen, eine tiefer gehende Diskussion hätte die ohnehin schon umfangreiche Ausarbeitung aber gesprengt.

Der vorgeschlagene Standard für taktgenaue busphasenbasierte *J-R*-Simulation mit TLM-2.0 eignet sich nachweislich für die Performance-Evaluation und für die Power-Analyse. Die Beispiele in dieser Arbeit beziehen sich meist auf die Performance-Evaluation, eine ausführliche Arbeit zum Thema Power-Analyse ist aber mit [Krue09] gegeben. Folgeforschungen können einerseits noch detailliertere Regeln für diese beiden sehr wichtigen Anwendungsfälle bestimmen oder aber auch noch andere Anwendungsfälle, wie die Low-Level-Protokoll-Verifikation diskutieren. Darüber hinaus kann auch untersucht werden, ob und wie die manuelle Neumodellierung der RTL-Modelle in taktgenauem TLM mit Hilfe von Entwurfsmustern

[1] Diese Erkenntnis wurde im Rahmen der Arbeiten an $[\text{OCP-09}]_i$ und $[\text{CaGr10}]_i$ gewonnen.
[2] AT, LT und taktgenaue Modellierung

oder -regeln für spezielle, immer wiederkehrende Modellierungssituationen[3] unterstützt oder gar automatisiert werden kann. Weiteres Potential zur Unterstützung der Designer liegt in der Definition der Busphasen. Diese werden im Rahmen meiner Arbeit manuell in Abhängigkeit vom Anwendungsfall bestimmt. Eine Suche nach einer noch einfacheren Notation und einer daraus folgenden Abbildung auf die Busphasen (wie sie schon in Abschnitt 6.2 skizziert wird) kann die Modellierung bisher nicht taktgenau TLM-modellierter memory-mapped Bus-Interfaces erleichtern. Denkbar sind auch Untersuchungen hinsichtlicher grafischer Busphasendefinitionen.

Es wird deutlich, dass dieser Standardisierungsvorschlag die Basis für ein großes Feld weiterer Forschung und Entwicklung legen kann und eine endgültige Standardisierung der taktgenauen TLM wichtig für den Fortschritt im Rahmen der taktgenauen, aber effizienten System-Modellierung ist.

[3] Z.B. Arbitrierung, Valid-Acknowledge-Handshakes, Timeouts oder Reset

Anhänge

A. Traits-Classes und Bindungschecks

Inhalt

A.1. Einleitung

In Abschnitt 2.3.2 wurde unter anderem der Mechanismus der Bindungschecks mit Hilfe von Types-Class (TC) erläutert. In diesem Anhang wird illustiert, wie TC einzusetzen sind und wie sie die korrekte Bindung sicher stellen.

A.2. Definieren einer neuen Traits-Class

Nehmen wir an, es gibt ein Protokoll, das im Grunde dem BP entspricht, aber zur korrekten Funktion ein eindeutige Identifikation des Senders benötigt. Nennen wir diese `MasterID`. Die `MasterID` kann dann als GP-Erweiterung wie in Listing A.1 gezeigt, definiert werden.

```
1 struct MasterID : public tlm::tlm_extension<MasterID>
2 {
3   unsigned int m_ID; //Die eigentliche ID
4   //Rest der Klasse
5   ...
6 };
```

Listing A.1: GP-Erweiterung `MasterID`

Da diese Erweiterung für die korrekte Funktion zwingend ist, wird eine neue TC definiert (siehe Listing A.2). Diese entspricht im Grunde der BP TC, wird aber dafür sorgen, dass ein direktes Binden eines Moduls mit `MasterID` an ein unerweitertes BP-Modul unmöglich ist.

```
1 struct base_protocol_with_master_id_types
2 {
3   typedef tlm_generic_payload tlm_payload_type;
4   typedef tlm_phase_type   tlm_phase_type;
5 };
```

Listing A.2: TC für erweitertes BP

A.3. Verwendung der Traits-Class für Bindungschecks

Nehmen wir nun ein Modul wie in Listing A.3 gezeigt. Dort wird ein Target-Modul definiert, dass mit dem um `MasterID` erweiterten BP betrieben werden soll; erkennbar daran, dass der Socket des Moduls die neue TC verwendet.

Dagegen stellt Listing A.4 ein Initiator-Modul dar, welches das unerweiterte BP nutzt, erkennbar daran, dass es die BP TC verwendet.

```
1  SC_MODULE(mid_target)
2  {
3    //Ein 32 Bit Target Socket, der das erweiterte BP verwendet
4    tlm::tlm_target_socket<32,base_protocol_with_master_id_types> socket;
5
6    SC_CTOR(mid_target): socket("socket")
7    {}
8
9    //Rest der Klasse
10   ...
11 };
```

Listing A.3: Target für erweitertes BP

```
1  SC_MODULE(bp_init)
2  {
3    //Ein 32 Bit Initiator Socket der das nicht erweiterte BP verwendet
4    tlm::tlm_initiator_socket<32,base_protocol_types> socket;
5
6    SC_CTOR(bp_init): socket("socket")
7    {}
8
9    //Rest der Klasse
10   ...
11 };
```

Listing A.4: Initiator für unerweitertes BP

Dadurch, dass die Sockets in Listings A.3 und A.4 unterschiedliche TC verwenden, können sie nicht direkt verbunden werden. Eine Implementierung von TLM-2.0 stellt sicher, dass zwei Sockets nur genau dann verbunden werden können, wenn ihre TC identisch sind. Folglich ist nun ein Adapter zwingend nötig.

A.4. Adapter zur Kopplung von Sockets mit unterschiedlichen Traits-Classes

Der Adapter ist in Listing A.5 zu sehen. Man erkennt, dass er einen Target-Socket für das unerweiterte BP besitzt, sodass der Initiator angebunden werden kann, und dass er einen Initiator-Socket für das erweiterte BP aufweist, damit das Target angebunden werden kann. Die notwendige Adaption ist in der Implementierung von `nb_transport_fw` bzw. `nb_transport_bw` zu sehen.

In `nb_transport_fw` wird lediglich eine `MasterID` zum Payload hinzugefügt, während in `nb_transport_bw` keine weiteren Maßnahmen nötig sind. Es wird deutlich, dass der Adapter

dank der Verwendung des Erweiterungsmechanismus sehr einfach und effizient bleibt. Bei einem GP ohne Erweiterungsmechanismus, müsste eine neue Payload-Klasse definiert werden und es bedürfte einer vollständigen Kopie des Inhaltes der Payloads innerhalb des Adapters.

```
class adapter : public sc_core::sc_module, ...
{
  //Ein 32 Bit Target Socket der das nicht erweiterte BP nutzt
  tlm:: tlm_target_socket<32,base_protocol_types> tsocket;

  //Ein 32 Bit Initiator Socket der das erweiterte BP nutzt
  tlm::tlm_initiator_socket<32,base_protocol_with_master_id_types> isocket;

  MasterID id;

  SC_CTOR(adapter): tsocket("tsocket"), isocket("isocket")
  {
    id.m_ID=get_unqiue_ID(); //Eindeutige Master ID erzeugen
  }

  //Wird von tsocket aufgerufen
  tlm::tlm_sync_enum
  nb_transport_fw(tlm::tlm_generic_payload& gp, tlm::tlm_phase& ph, sc_core::sc_time& time)
  {
    gp.set_extension(&id); //Die eindeutige Master ID dem Payload hinzufuegen
    //...dann einfach den Aufruf an das eigentliche Ziel weiterleiten
    return isocket->nb_transport_fw(gp,ph,time);
  }

  //Wird von isocket aufgerufen
  tlm::tlm_sync_enum
  nb_transport_bw(tlm::tlm_generic_payload& gp, tlm::tlm_phase& ph, sc_core::sc_time& time)
  {
    //Den Aufruf einfach vom isocket an den tsocket weiterreichen
    return tsocket->nb_transport_bw(gp,ph,time);
  }

  //Rest der Klasse
  ...
};
```

Listing A.5: Adapter für `MasterID`

Nun kann eine Verbindung zwischen Initiator und Target, wie in Listing A.6 gezeigt, aufgebaut werden.

```
bp_init initiator("initiator"); //Initiator erzeugen
mid_target target("target"); //Target erzeugen
adapter adapt("adapt"); //Adapter erzeugen

initiator.socket(adapt.tsocket); //Verbinde Initiator Socket des Initiators
                                 //  mit Target Socket des Adapters
adapt.isocket(target.socket); //Verbinde Initiator Socket des Adapters
                                 //  mit Target Socket des Targets
```

Listing A.6: Verbindung von erweitertem und unerweitertem BP über einen Adapter

B. Über Busphasen

Inhalt

B.1. Einleitung

Zur Festlegung der J-R-Simulation zum Zwecke der Performance-Simulation habe ich diverse MMBIF ([ARM-03, ARM-99, IBM-01, IBM-04, STM-07], [Herv02, OCP-08]$_i$) und eine Vielzahl proprietärer Interfaces ([Bode07, Günz08]) untersucht bzw. untersuchen lassen. Dieser Anhang listet die Ergebnisse der Untersuchungen und fasst die daraus gezogenen Schlüsse zusammen.

B.2. Phasen

Als Bus- bzw. Kommunikationsphase bezeichne ich die Übergabe eines Datensatzes von einem Sender (Signaltreiber) an einen Empfänger (Signalsenke), wobei der Empfänger eine Antwort zurückgeben kann. Ein Datensatz ist dabei eine beliebige Menge von RTL-Signalen, die vom Sender zum Empfänger verlaufen, während eine Antwort ebenso eine Menge von RTL-Signalen ist, deren Richtung aber invers zu der des Datensatzes ist.

Eine Busphase hat die folgenden Eigenschaften:

1. Der Start der Phase wird durch den Zustand der Signale des Senders im aktuellen Takt und/oder durch den Zustand von beliebigen Signalen in vergangenen Takten bestimmt.
2. Der Datensatz bleibt nach dem Start und vor dem Ende bzw. Abbruch der Phase stabil.
3. Die Antwort ist nur genau am Phasenende gültig, wenn die Phase nicht abgebrochen wurde. Vorher sind exakte Werte der Antwort ohne Bedeutung. Die Phase muss lediglich als nicht beendet identifizierbar sein.
4. Die exakten Werte von Datensatz und Antwort sind zwischen Ende und Start der Phase ohne Bedeutung. Die Phase muss lediglich als nicht gestartet identifizierbar sein.

Das bedeutet, dass sich die Signale nur zu genau zwei Zeitpunkten ändern können: Wenn die Phase beginnt oder wenn sie endet bzw. abbricht. Es ist wichtig festzustellen, dass dies nicht heißt das bei einem gegebenen MMBIF Änderungen zwischen diesen Zeitpunkten auf den Signalen einer Busphase ausgeschlossen sind. Sie werden aber in der busphasenbasierten Simulation dann unterdrückt; es wird also von diesen Änderungen abstrahiert (Siehe auch Anhang D). Dies kann aus verschiedenen Gründen wünschenswert sein. Im Rahmen der Performance-Evaluation zum Beispiel ist es nicht notwendig gewisse Protokollfehler, wie z.B. das Ändern der Adresse während der Requestphase auf dem PLB, zu modellieren. Oder aber man kann mit geschickter Definition von Busphasen auch Signaländerungen unterdrücken (im Modell, wohl gemerkt), die auf die Kommunikation keine Auswirkung haben, wie zum Beispiel Änderungen in Acknowledge-Signalen (z.B. SCmdAccept im OCP) während gar keine Kommunikation stattfindet (im Beispiel also gar kein MCmd im OCP gesetzt ist).

Im Folgenden werden nun die häufigsten von mir bei der Analyse der MMBIF und proprietären Interfaces identifizierten Kommunikationsabläufe aufgelistet und es wird jeder auf Busphasen abgebildet, die die oben genannten Eigenschaften erfüllen. In den Abbildungen sind die Signale immer nach ihrem Treiber benannt. Das heißt zum Beispiel, dass ein Signal vom Sender zum Empfänger das Präfix „Sender_“ erhält.

B.2.1. Infinite Phase

Die infinite Phase zeichnet sich dadurch aus, dass nur der Sender Signale treibt. Die Signale werden nie explizit als gültig markiert, sodass der Empfänger diese stets als gültig ansehen muss und so die Kommunikation im Grunde nie endet. Dementsprechend sind die Signale niemals in einem „don't-care“-Zustand und es gibt für diese Phase keinen Endzeitpunkt, aber mehrere Startzeitpunkte. Mit einem neuen Start der Phase, endet sie zwar auch, jedoch gibt es nie einen Takt in dem die Phase nicht aktiv ist. Die Phase startet, wann immer sich ein Signal des Datensatzes ändert. Für die Ports von infiniten Phasen kann eine beliebiger Signalwert als Default- bzw. Initial-Zustand ausgewählt werden, sodass die infinite Phase zum ersten Mal startet, wenn ein Port der Phase diesen Zustand verläßt. Im Beispiel in Abbildung B.1 soll der Default-Zustand der Wert in Takt 1 sein und somit sind Startzeitpunkte in den Takten 2, 3, 5, 6, 7, 11 und 13.

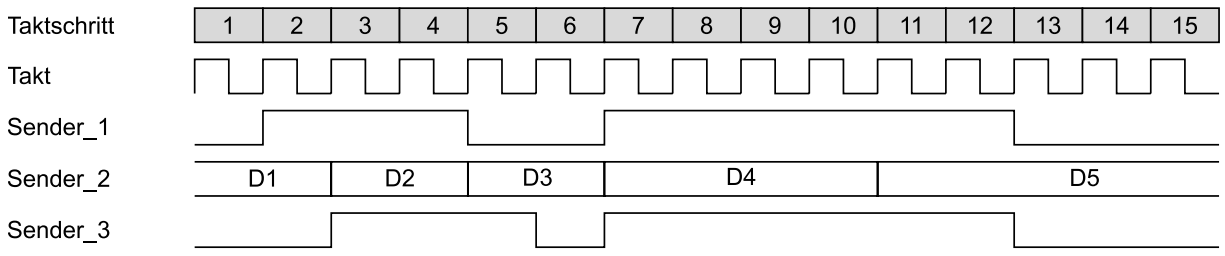


Abbildung B.1.: Infinite Busphase

Mit einer Signaländerung bestimmt der Sender den Start der Phase (Eigenschaft 1). Eine Änderung des Datensatzes beendet die Phase und startet sie erneut, folglich trifft auch Eigenschaft 2 zu. Da es weder eine Antwort noch Takte zwischen Phasenende und -start gibt, treffen auch die Eigenschaften 3 und 4 zu. Ein Beispiel für eine solche Kommunikation ist die Übergabe an Daten an einen arithmetischen Co-Prozessor mit fester Latenz.

B.2.2. Einzeltaktphase

Auch bei der Einzeltaktphase sind nur Signale des Senders beteiligt. Im Gegensatz zur infiniten Phase gibt es aber ein oder mehr Signale des Datensatzes, die die Gültigkeit der Phase markieren können. Nehmen diese Signale oder dieses Signal eine von potentiell mehreren vordefinierten Kombinationen an, so gilt die Phase als gültig. Ist die Phase nicht gültig, muss der Empfänger die Signale des Datensatzes ignorieren. Die Kommunikation dauert exakt einen Takt, endet also im gleichen Takt in dem sie startet. Bleibt die Phase mehr als einen Takt gültig, so wird mehrfach kommuniziert. Im Beispiel in Abbildung B.2 markiert das Signal Sender_Valid bei einem Wert von logisch Eins die Phase als gültig. Somit startet und endet die Kommunikation in den Takten 2, 4, 9, 10 und 11.

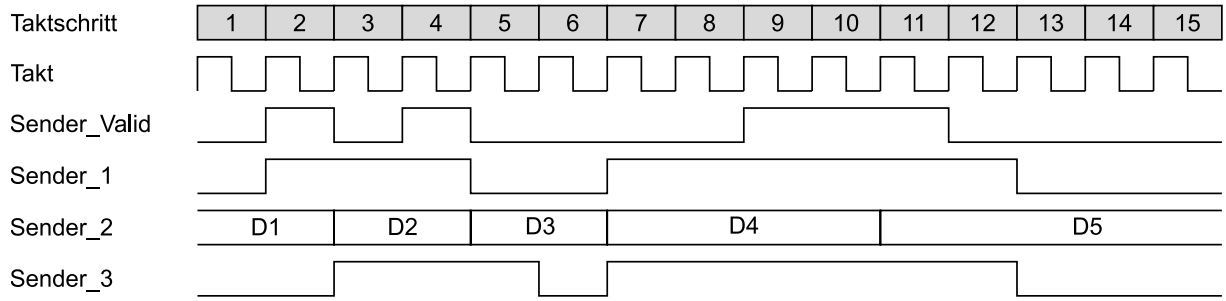


Abbildung B.2.: Einzeltakt-Busphase

Der Sender startet die Phase mit dem Setzen einer entsprechenden Signalkombination auf den Ports, die die Gültigkeit anzeigen können (Eigenschaft 1). Eigenschaft 2 ist erfüllt, da die Phase exakt einen Takt dauert, während Eigenschaft 3 aufgrund der Abwesenheit einer Antwort erfüllt wird. Basierend auf der oben genannten Bedeutung der Gültigkeit, wird auch Eigenschaft 4 erfüllt. Beispiele für eine solche Phase sind der Slave-seitige Addresszyklus im AMBA AHB oder das Schreiben in einen FIFO.

B.2.3. Multitaktphase

Bei der Multitaktphase verwendet der Sender, genau wie bei der Einzeltaktphase, Signale des Datensatzes, um die Gültigkeit des Datensatzes anzuzeigen. Im Gegensatz zur Einzeltaktphase muss der Empfänger den Datensatz nicht im gleichen Takt übernehmen. Er zeigt die Übernahme durch das Setzen eines oder mehrerer vordefinierter Bestätigungssignalkombinationen an. Das bedeutet, dass mindestens ein Signal als Antwort vom Empfänger zum Sender

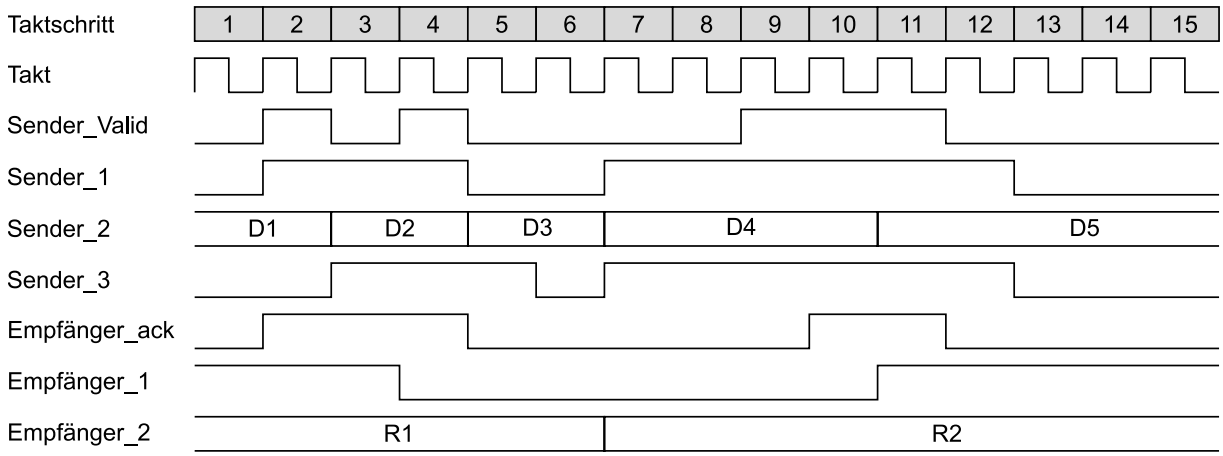


Abbildung B.3.: Multitakt-Busphase

verläuft, jedoch kann die Antwort auch noch mehr Signale enthalten (z.B. angeforderte Lesedaten). Die Phase endet in dem Takt, in dem die Datenübernahme bestätigt wurde. Markiert der Sender den Datensatz als nicht gültig, bevor der Empfänger den Empfang bestätigt, so gilt die Phase als abgebrochen. Sie endet dann in dem Takt, in dem die Ungültigkeit der Phase identifiziert wird. Solch ein Abbruch ist eine Entartung eines Starts, da der Abbruch genau wie der Start einzig vom Sender angezeigt werden kann und dabei exakt die gleichen Signale verwendet werden. Darüber hinaus ist ein Abbruch in allen von mir untersuchten Protokollen stärker als eine Bestätigung. Das heißt, bestätigt der Empfänger im gleichen Takt in dem der Sender abbricht, so gilt die Phase als abgebrochen.

Die oben erwähnte Markierung der Gültigkeit wird nahezu immer vom Sender selbst vorgenommen. Ich habe aber noch eine besondere Form der Gültigkeitsmarkierung gefunden: Der Start einer Phase kann auch von einer speziellen Signalkombination in der Vergangenheit (von nicht notwendigerweise zum Datensatz oder zur Antwort der Phase gehörenden Signalen) erzwungen oder unterdrückt werden, wenn z.B. das Ende einer Phase den Start einer anderen Phase bedingt.

Der Sender startet eine Phase, indem sie als gültig markiert wird oder aber die Phase startet aufgrund einer speziellen Antwort in einem vorangegangenen Takt (Eigenschaft 1). Da der Sender erst weiß, wann der Empfänger die Daten übernimmt, wenn dieser das anzeigt, muss der Sender die Daten bis zu diesem Zeitpunkt stabil halten; eine Änderung des Datensatzes ist nur erlaubt, wenn dies die Phase abbricht und so die Änderung mit dem Ende der Phase zusammen fällt (Eigenschaft 2). Antwortdaten kann der Sender erst als gültig identifizieren, wenn die Übernahme bestätigt wird, da es sich vorher um Zwischenergebnisse bei der Berechnung der Antwort handeln kann; die Antwort ist nur zusammen mit einem gültigen Datensatz aussagekräftig (Eigenschaft 3). Vor dem Start und nach dem Ende der Phase sind die exakten Werte der Signale nicht relevant, einzig die Tatsache, dass die Phase nicht gültig ist, ist entscheident (Eigenschaft 4).

Im Beispiel in Abbildung B.3 zeigt der Empfänger die Übernahme des Datensatzes durch Setzen des Signales Empfänger_ack auf logisch Eins an. Die Gültigkeit wird durch Sender_Valid markiert. Somit startet die Phase in den Takten 2, 4, 9 und 11 und endet in den

Takten 2, 4, 10 und 11. Dementsprechend ist der Datensatz des Senders nur in den Takten 2, 4, 9, 10 und 11 gültig, während die Antwort des Empfängers nur in den Takten 2, 4, 10 und 11 gültig ist.

Beispiele für diese Art der Phase sind alle OCP-Phasen bei Verwendung der OCP-Accept-Signalisierung, PLB-Request- und PLB-Datenphase, OPB-Transferphase, alle AXI-Phasen, AHB-Datenphase und APB-Transferphase.

B.2.4. Phasen mit Erlaubnis

Bei der Phase mit Erlaubnis treibt auch der Empfänger mindestens ein Signal. Nimmt dieses Signal (oder diese Signale) einen Wert aus einer vordefinierten Menge von Signalkombinationen an, so zeigt der Empfänger damit an, dass er zur Kommunikation bereit ist. Ist er nicht zur Kommunikation bereit, ignoriert er alle Signale des Senders. Zusätzlich kann der Sender Signale des Datensatzes nutzen, um die Gültigkeit des Datensatzes anzuzeigen (vergleiche Abschnitt B.2.2) und in diesem Fall kann auch wiederum der Empfänger die Phase quittieren (vergleiche Abschnitt B.2.3). Alle bisher eingeführten Phasen können also noch mit einer Erlaubnis versehen werden.

Das Signal (oder die Signale), das die Empfangsbereitschaft anzeigt, ist grundsätzlich vom aktuellen Zustand des Senderdatensatzes entkoppelt. Es wird vom Sender verwendet, um zu entscheiden, ob eine Kommunikation starten kann oder nicht, kann also nicht selbst davon abhängen ob eine Kommunikation gestartet wird[1]. Somit handelt es sich bei der Phase mit Erlaubnis, um zwei getrennte Kommunikationen.

Verwendet der Sender keine explizite Gültigkeitssignalisierung (Abbildung B.4a), so gibt es eine infinite Phase die vom Empfänger zum Sender geht (es liegt also ein Rollentausch vor). Der Datensatz dieser Phase sind die Signale, die die Empfangsbereitschaft signalisieren. Dazu gibt es eine weitere infinite Phase vom Sender zum Empfänger, mit der der eigentliche Datensatz übertragen wird. Die Einschränkung dabei ist, dass diese Phase aber nur dann startet, wenn die erste Phase eine Empfangsbereitschaft im aktuellen Zyklus signalisiert hat. Es handelt sich also um ein den Phasen übergeordnetes Kommunikationsprotokoll. In Abbildung B.4a bedeutet dies also, dass die infinite Empfangsbereitschaftsphase in den Takten 3 und 11 startet, während die infinite Datensatzübertragungsphase nur in den Takten 3, 5, 6 und 7 startet (vgl. Abschnitt B.2.1). Die übrigen Änderungen des Datensatzes (und somit eigentlich Starts der infiniten Phase vom Sender zum Empfänger) sind ohne Bedeutung.

Verwendet der Sender dagegen explizite Gültigkeitssignalisierung (Abbildung B.4b und c), so gibt es auch wieder eine infinite Phase die vom Empfänger zum Sender geht. Der Datensatz dieser Phase sind die Signale, die die Empfangsbereitschaft signalisieren. Dagegen gibt es eine Einzeltaktphase (Abbildung B.4b) oder eine Multitaktphase (Abbildung B.4c) vom Sender zum Empfänger, mit der der eigentliche Datensatz (und ggf. die Antwort) übertragen wird. Die Einschränkung dabei ist, dass diese Phase aber nur dann startet und endet, wenn die

[1] Es entstünde ein kombinatorischer Zyklus. Im schlimmsten Falle eine astabile Kippstufe.

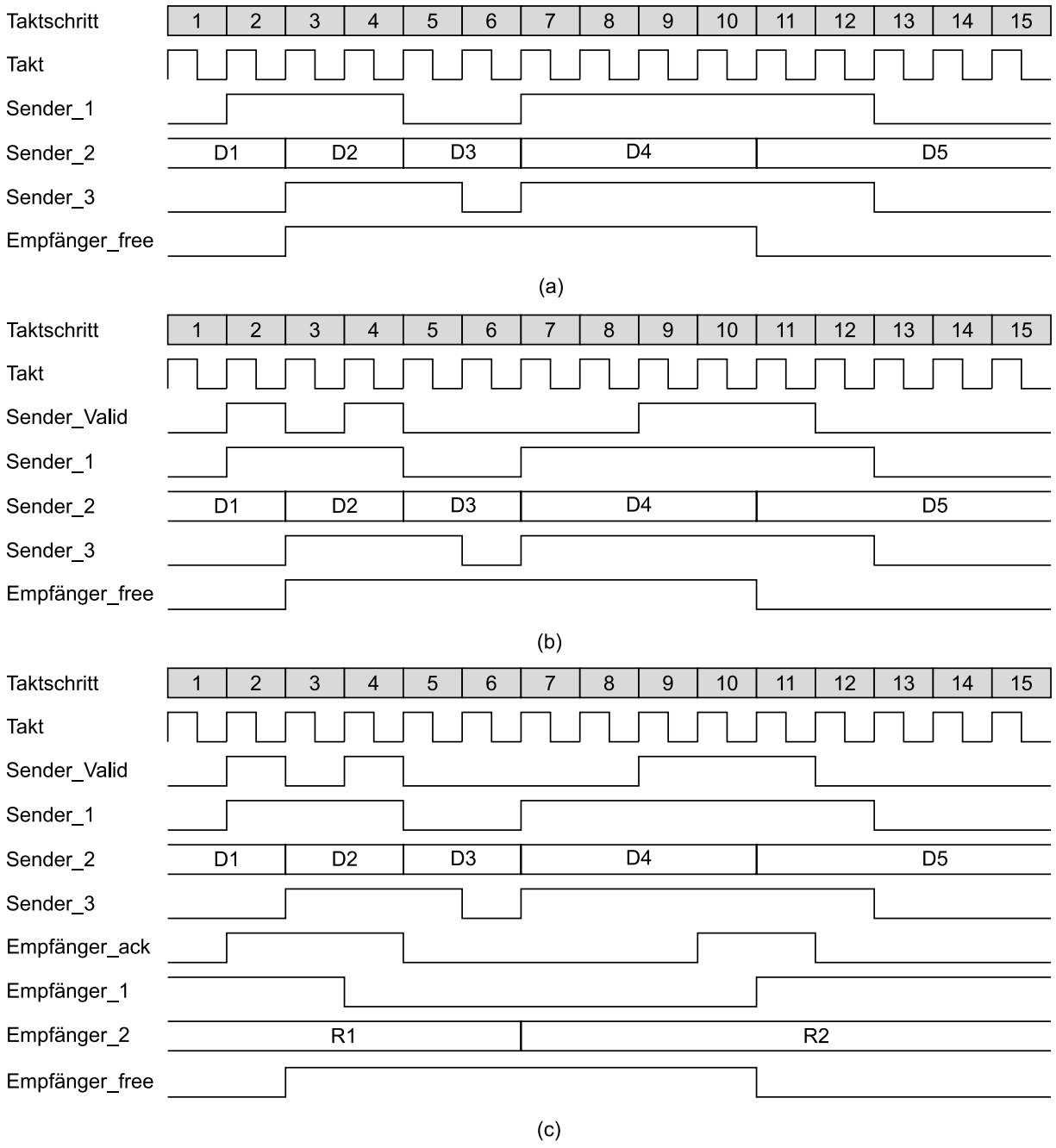


Abbildung B.4.: Phasen mit Erlaubnis

erste Phase eine Empfangsbereitschaft im aktuellen Zyklus signalisiert hat. In Abbildung B.4b bedeutet dies also, dass die infinite Empfangsbereitschaftsphase in den Takten 3 und 11 startet, während die Einzeltakt-Datensatzübertragungsphase in den Takten 4, 9 und 10 startet (vergleiche Abschnitt B.2.2), während in Abbildung B.4c die Multitaktphase nur noch in den Takten 4 und 9 startet und in den Takten 4 und 10 endet (vgl. Abschnitt B.2.3)[2].

Damit diese Einschränkungen der Phasenstarts und gegebenenfalls auch Phasenenden mit Hilfe von anderen Signalen, die nicht zum Datensatz bzw. Antwort der Phase gehören, möglich ist, muss eine Phase also neben Datensatz und Antwort auch noch die Menge der Signale

[2]In einer „echten“ Kommunikation hätte der Sender die Daten ohne Empfangsbereitschaft nicht als gültig markiert. Dies geschieht hier zur Hervorhebung des Unterschieds zu Phasen ohne Erlaubnis.

kennen, die ihre Starts und Enden beeinflusst, ohne dass die Phase selbst diese Werte beeinflussen kann. In Abbildung B.4, wäre dieses Signal Empfänger_free. Starten kann die Phase nur, wenn das Signal auf Eins ist, jedoch kann der Sender dieses Signal nicht selbst setzen um die Phasen zu starten. Darüber hinaus kann die Phase auch nicht die Stabilität dieses Signales erzwingen, da es an eine andere Phase gekoppelt ist. Dieser zusätzliche Signalsatz muss also Teil einer Phasenbeschreibung sein.

Die infinite Phase, die Einzeltakt- und Multitaktphase erfüllen die Grundeigenschaften von Busphasen. Da die Phase mit Erlaubnis auf diese drei zurückgeführt wurde, werden auch in diesem Fall die Eigenschaften erfüllt. Beispiele für solche Phasen sind OCP-Request-, OCP-Data- oder OCP-Responsephase bei Verwendung der OCP-Busy-Signalisierung oder Schreiben in einen FIFO der ein FIFO-Voll-Signal hat.

B.2.5. Anmerkungen

Bei der Untersuchung der verschiedenen Interfaces habe ich keines gefunden, dessen Kommunikation nicht auf die oben genannten Phasen abgebildet werden kann. Im Zweifelsfalle kann immer für jedes einzelne Signal im Interface eine infinite Phase definiert werden, was aber zu einer großen Menge an Phasen und im Extremfall zu einer RTL-Simulation führt. Es sollte stets versucht werden finite Phasen mit größtmöglichen Daten- und Antwortsätzen zu finden. Es können auch Phasentypen verwendet werden, die hier nicht gelistet sind. Wichtig bei der Identifizierung von Phasen ist es, sicher zu stellen, dass die Phasen die 4 zentralen Eigenschaften erfüllen.

Als Beispiel wie dies getan werden kann, soll die Request-Phase des PLB dienen. Ihre Beschreibung in der PLB-Spezifikation legt nahe diese direkt auf eine Multitaktphase abzubilden, jedoch darf der Sender während der Request als gültig markiert ist und er auf die Bestätigung wartet, die Priorität des Requests ändern. Dies verletzt Eigenschaft 2. Um dies zu vermeiden, kann man entweder die Priorität aus der Phase entfernen und dafür eine eigene infinite Phase definieren, oder aber man definiert, dass mit Änderung der Priorität die aktuelle Requestphase mit neuer Priorität erneut beginnt (vgl. wiederholter Start einer infiniten Phase). Dann werden wieder alle Eigenschaften erfüllt.

Das Finden eines Default-Zustandes für Ports infiniter Phasen wurde bereits in Abschnitt B.2.1 erläutert. Das Finden eines Default-Zustandes für finite Phasen ist grundsätzlich dadurch möglich eine Signalkombination zu wählen, die nicht die Gültigkeit anzeigt. Wird eine Folge länger als einen Takt zur Anzeige der Gültigkeit verwendet, kann theoretisch jede Signalkombination zumindest in einer Folge vorkommen. Zum Finden eines statischen Defaultzustandes muss dann eine Kombination gewählt werden, die kontinuierlich wiederholt keine Gültigkeit anzeigt. Nun ist es wiederum möglich, dass das kontinuierliche Wiederholen jeder beliebigen Signalkombination gerade die Gültigkeit anzeigt. In diesem Fall ist es nicht möglich einen statischen Defaulzustand zu definieren. Die Anzeige der Ungültigkeit der Phase muss dann über ein taktweise wechselnden Signalfolge realisiert werden. Dieser Fall wird in der vorliegenden Arbeit vorsätzlich ignoriert, da das Anzeigen der Ungültigkeit

einer Phase durch dauerhaft Signalwechsel aus Sicht der Energieeffizienz unvernünftig ist und somit in realen Schaltungen nicht verwendet werden wird. Das Phasenmodell und insbesondere die Defaultzustände basierend auf dieser rein theoretischen Möglichkeit unnötig zu verkomplizieren, ist nicht sinnvoll.

Des Weiteren können sich Phasen sowohl zeitlich als auch räumlich (also in der Menge der von ihnen verwendeten Ports) überlappen. Zeitliche Überlappungen sind ohne räumliche Überlappungen unkritisch (Abbildung B.5, Phasen A und B). Gleiches gilt für räumliche Überlappungen, wenn zeitliche Überlappungen durch das darüberliegende Protokoll ausgeschlossen werden (Abbildung B.5, Phasen C und D). Problematisch sind Phasen die sich zeitlich und räumlich überlappen (Abbildung B.5, Phasen E und F). Startet Phase F zwischen Start und Ende von Phase E, so kann dies potentiell Änderungen in den überlappenden Signalen bewirken und so Eigenschaft 2 von Phase E verletzen. Eine solche Überlappung kann aufgelöst werden, indem die überlappenden Signale in einer neuen, infiniten Phase (Abbildung B.5, Phase G und reduzierte Phasen E und F) zusammengefasst werden und man in einem darüberliegenden Protokoll Abhängigkeiten zwischen den Phasen definiert. Jedoch habe ich kein Protokoll finden können, bei dem dieser Fall eintritt. Das einzige, von mir untersuchte Protokoll bei dem sich Phasen räumlich und zeitlich überlappen, ist das AMBA-AHB-Protokoll. Dort überlappen die Master-seitige Address- und Datenphase einzig im verwendeten Bestätigungssignal. Da beide Phasen den gleichen Sender haben (Master) bedeutet das, dass die Signale, die sich beim Start der beiden Phasen ändern können nicht überlappen. Ändert sich das überlappende Signal, d.h. eine Phase endet, so legt das AHB-Protokoll fest, dass auch die andere Phase endet (die Änderung also keine Phaseneigenschaft verletzt). Die Überlappung ist in diesem Fall also unkritisch.

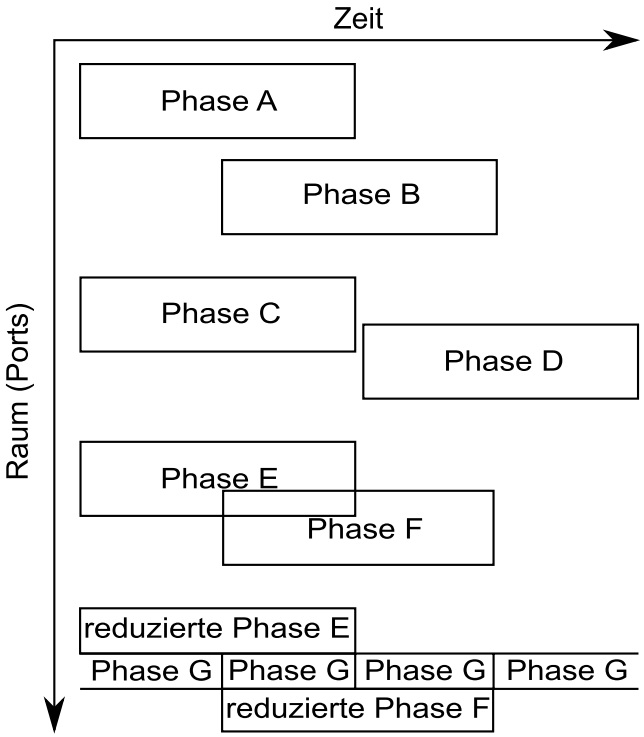


Abbildung B.5.: Überlappungen bei Busphasen

Die Zulässigkeit der räumlichen und zeitlichen Überlappung ist also nur gegeben, wenn sichergestellt ist, dass die Überlappung die Eigenschaften der Phase nicht gefährden. Ansonsten muss auf die oben erwähnte Auflösung der räumlichen Überlappung zurückgegriffen werden.

B.3. Fazit

Um die oben beschriebenen Busphasen in einer Beschreibung zu erfassen, müssen die folgenden Daten in der Beschreibung erfasst werden:

- Die Menge von Signalen, die beim Start oder Abbruch der Phase gesetzt bzw. berechnet werden müssen. Dies ist grundsätzlich immer der Datensatz zuzüglich der Signale der Antwort, die das Ende anzeigen können. Letztere müssen beim Start auch gesetzt bzw. berechnet werden, damit klar ist, dass die Phase nicht sofort endet. Endet sie sofort, müssen auch die im folgenden Punkt genannten Signale berechnet werden.
- Die eventuell leere Menge von Signalen, die beim Ende der Phase gesetzt bzw. berechnet werden müssen. Diese Menge entspricht grundsätzlich den Signalen aus der Antwort der Phase.
- Die eventuell leere Menge von Signalen, die nicht zum Datensatz oder zur Antwort der Phase gehören, jedoch den Start, den Abbruch oder das Ende der Phase beeinflussen können. Solche Signale sind z. B. Erlaubnissignale (siehe Abschnitt B.2.4) oder Signale deren vergangene Werte einen Start der Phase erzwingen oder unterdrücken können (siehe Abschnitt B.2.3).
- Eine Beschreibung unter welchen Bedingungen die Phase startet.
- Eine Beschreibung unter welchen Bedingungen die Phase abbricht.
- Eine Beschreibung unter welchen Bedingungen die Phase endet.

Eine solche Beschreibung wird in Abschnitt 3.2.1 Definition 3.7 eingeführt.

C. Bezüglich mehrerer Taktdomänen

Inhalt

C.1. Einleitung

Wie in Abschnitt 3.1 erläutert, betrachte ich die Indizes an Input- und Output-Abläufen als Takte einer RTL-Simulation. Folglich kann eine Kommunikationsstruktur nur einen Takt verwenden. Nun existieren jedoch diverse SoCs in denen mehr als ein Takt existiert und in denen Module aus verschiedenen Taktdomänen miteinander kommunizieren. Dieser Abschnitt geht darauf ein, wie solche Systeme auch mit den in Abschnitt 3.1 definierten Kommunikationsstrukturen erfasst werden können.

C.2. Taktdomänenübergang

Abbildung C.1 zeigt den allgemeinen Fall bei der Existenz mehrerer Takte: Ein Modul (im Beispiel ein Master) kommuniziert mit einem Kommunikationsautomat, dabei haben beide unterschiedliche Takte[1].

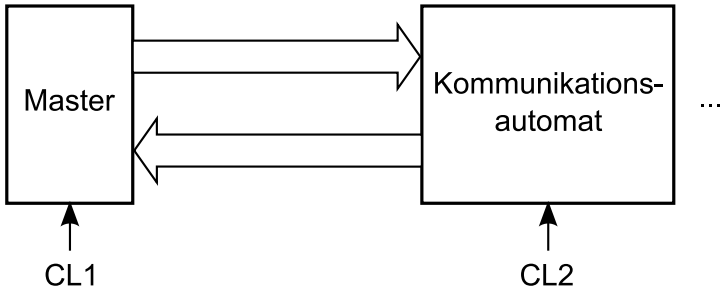


Abbildung C.1.: Master und Kommunikationsautomat mit unterschiedlichen Takten

[1]Hat ein Kommunikationsmedium mehr als einen Takteingang, so können bei allen von mir untersuchten Kommunikationsmedien durch Absteigen in der Modulhierarchie, also als Submodule, Module gefunden werden, die nur durch einen einzigen Takt gespeist werden. Auf dieser Ebene muss dann die Festlegung von Kommunikationsstrukturen stattfinden.

Damit eine in Abbildung C.1 gezeigte Verbindung korrekt arbeiten kann, muss ein synchronisierter Übergang zwischen den unterschiedlichen Taktdomänen existieren. Dieser Übergang existiert entweder in Form von zwischen den Modulen vorhandenen Synchronisierern (Abbildung C.2a) oder innerhalb der Module selbst (Abbildung C.2b)[2]. Findet die Synchonisierung innerhalb des Moduls statt kann aber ein (eventuell nicht explizit so vorhandenes) Submodul des Masters bzw. Kommunikationsautomaten identifiziert werden, dass ausschließlich mit synchronisierten Werten arbeitet. Man erkennt, dass dann abgesehen von den Modulgrenzen die Abbildungen C.2a) und C.2b) gleich sind.

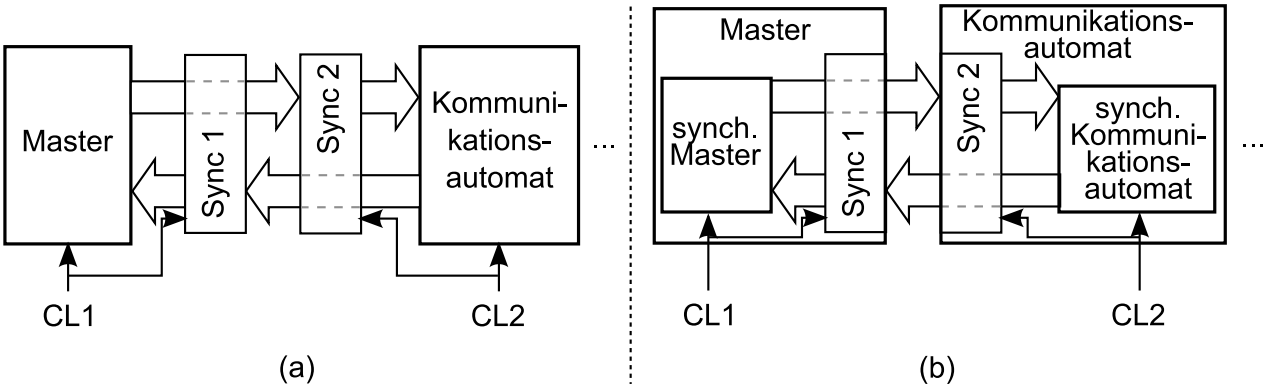


Abbildung C.2.: Master und Kommunikationsautomat mit unterschiedlichen Takten und Synchronisierern

Ist der Takt des Kommunikationsautomaten (CL2) schneller als der Takt des Masters (CL1), so gibt es bei der Definition der Kommunikationsstruktur keine Probleme, weil in einem Takt des Kommunikationsautomaten immer auch nur ein Input vom Master erzeugt werden kann. Ist der Takt des Masters schneller als der Takt des Kommunikationsautomaten, so kann der Master (zumindest theoretisch) mehrere verschiedene Inputs in einem CL2 Takt übergeben. Da diese Inputs aber synchronisiert werden, wird ohnehin nur der letzte übergebene Input im entsprechenden Takt wirksam. Dieser Input ist derjenige der im entsprechenden Inputablauf berücksichtigt wird, da die anderen Inputs im gleichen Takt, genau wie mehrfache Änderung innerhalb eines Deltazyklus, für den Kommunikationsautomaten „unsichtbar“ sind.

Somit können auch bei Mastern und Slaves, die eine andere Taktfrequenz als der Kommunikationsautomat haben, die Annahme aufrecht erhalten werden, dass die Indizes an den Input- und Output-Abläufen die Takte des Kommunikationsautomaten sind.

[2]Auch hier kann die gezeigte Struktur durch Absteigen in der Modulhierarchie gefunden. Existiert z.B. zwischen einem Master und einem Kommunikationsmedium eine synchronisierte Kommunikation und parallel dazu eine Kommunikation die ohne Synchronisation das Kommunikationsmedium passiert, so ist dieses Kommunikationsmedium in zwei Kommunikationsautomaten zu zerlegen.: Einen, der einen Synchronisierer und einen eigenen Takt hat und einen, der mit dem Takt des Masters arbeitet.

D. Beispiel zur taktgenauen busphasenbasierten J-R-Simulation: OPB

Inhalt

D.1. Einleitung

Kapitel 3 führt die Begriffe der Input-Abstraktion, der Relevanzauswahl und der darauf basierenden J-R-Simulation ein. Desweiteren zeigt Kapitel 3, wie mit Hilfe relativ einfacher Busphasen, die Input-Abstraktion und die Relevanzauswahl definiert werden können. Dies geschieht in einem mathematischen Modell und soll in diesem Anhang an einem konkreten Beispiel verdeutlicht werden. Dabei wird nicht die eigentliche Simulation des OPB gezeigt, sondern vielmehr der Effekt den Input-Abstraktion und Relevanzauswahl haben und warum eine darauf angepasste J-R-Simulation effizienter sein kann, als ein taktgenaue Simulation.

D.2. On-Chip Peripheral Bus

Der On-Chip-Peripheral-Bus (OPB) [IBM-01] ist Teil der IBM CoreConnect Busarchitektur. Der OPB wurde mit dem Ziel entwickelt, IPs zu verbinden, die hauptsächlich über Einzelworttransfers oder nur kurze Multiworttransfers kommunizieren. Dies sind in der Regel kleine Block-RAMs oder IP-Blöcke, die ein memory-mapped Registerinterface haben, wie z.B. UART-Controller, Keyboard-Contoller etc. Er ist nicht dafür gedacht, dass Prozessoren direkt daran angeschlossen werden, da diese deutlich höhere Anforderungen bezüglich des Datendurchsatzes stellen. Prozessoren können über eine Busbrücke vom Prozessorbus aus

auf den OPB zugreifen. In diesem Anhang werden die Konzepte aus Abschnitt 3.2 anhand des OPB verdeutlicht. Das Protokoll des OPB wird hier nicht im Detail erläutert, dazu sei auf die angegebene Sekundärliteratur verwiesen.

D.3. Phasen im OPB-Master-Interface

Die taktgenaue busphasenbasierte *J-R*-Simulation wird in diesem Anhang am Beispiel der Master-Schnittstelle eines Masters am OPB dargestellt. Dementsprechend werden die zur Festlegung von J und R notwendigen Busphasen für einen Master nun definiert.

Es sei ein 32-Bit OPB mit dem Kommunikationsautomat $KA = (Z, z_0, I, O, \delta, \lambda)$ und der Peripherie $Per = (M, S, PN, P, pz)$ gegeben. Es sei $m1 \in M$ ein Master aus der Peripherie Per des OPB. Dabei gilt

$$pz(m1) = \left\{ \begin{array}{l} (M1\_request, 1, e), (M1\_busLock, 1, e), (M1\_select, 1, e), \\ (M1\_DBus, 32, e), (M1\_BE, 4, e), (M1\_beXfer, 1, e), \\ (M1\_hwXfer, 1, e), (M1\_fwXfer, 1, e), (M1\_RNW, 1, e), \\ (M1\_seqAddr, 1, e), (M1\_DBusEn, 1, e), (M1\_ABus, 32, e), \\ (OPB\_M1Grant, 1, a), (OPB\_xferAck, 1, a), (OPB\_errAck, 1, a), \\ (OPB\_beAck, 1, a), (OPB\_hwAck, 1, a), (OPB\_fwAck, 1, a), \\ (OPB\_retry, 1, a), (OPB\_timeout, 1, a), (OPB\_DBus, 32, a) \end{array} \right\}$$

Die Analyse der OPB-Spezifikation zeigt, dass ein Master vor einem Transfer den Bus anfordern muss. Dies tut er mit dem Signal `M1_request`. Keine weitere Aktivität des Masters ist notwendig. Der Master kann den Bus in jedem beliebigen Takt anfordern und die Anforderung auch wieder verwerfen. Es bietet sich hier die Verwendung einer infiniten Phase an (siehe Anhang B), welche ich $busreqM1$ nenne. Die Menge der Ports dieser Phase besteht einzig aus $PP_{busreqM1} = \{(M1\_request, 1, e)\}$. Für die formale Erfassung der Busphase ergeben sich damit die Phasenstartports und Phasenendports zu $PP_{busreqM1S} = PP_{busreqM1}, PP_{busreqM1E} = \emptyset$ und die Beobachtungsdauer zu $tb_{busreqM1} = 2$. Keine andere Phase hat Einfluss auf die Starts von $busreqM1$, also kann $PP_{busreqM1O} = \emptyset$ gesetzt werden. Da es eine infinite Phase ist bleiben End- und Abbruchkriterium leer. Das Startkriterium $S_{busreqM1} = \left\{ \; (w1, w2) \in (SI_{(M1\_request,1,e)})^2 \;\middle|\; w1 \neq w2 \; \right\}$ ist erfüllt, sobald sich das Signal ändert, genau wie für eine infinite Phase gewünscht.

Der Bus erteilt dem Master mit dem Signal `OPB_M1Grant` den Zugriff auf den Bus. Dies kann auch ohne eine Anforderung seitens des Masters passieren[1]. Auch hier soll wieder eine infinite Phase namens $grantM1$ verwendet werden, welche sich analog zum Vorgehen für $busreqM1$ ergibt.

Ein Master kann die Arbitrierung des Busses blockieren, indem er das Signal `M1_busLock` verwendet. Er kann dies zu jedem beliebigen Zeitpunkt tun, vorausgesetzt der Bus hat ihm Zugriff erteilt. Danach ist die Aufhebung an keinerlei anderes Signal oder eine andere Phase

[1] Aus diesem Grund sind die Phasen für die Anforderung und die Zuteilung auch unabhängig

gebunden. Dementsprechend wird auch für die Phase $lockM1$ eine infinite Phasen mit der Portmenge $PP_{lockM1} = \{(M1\_busLock, 1, e)\}$ verwendet.

Ein Transfer wird von einem Master über das Setzen des Signals `M1_select` gestartet. Er endet wenn der Master zu dem `M1_select`-Signal `OPB_xferAck`, `OPB_timeout` oder `OPB_retry` empfängt. Der Master kann den Transfer abbrechen, indem er das `M1_select`-Signal wieder auf Null setzt. Es handelt sich also um eine klassische Multitaktphase, welche ich $xferM1$ nenne. Zur Straffung der folgenden Notation werden die Menge der Ausgangssignale des Masters, die noch keiner Phase zugeordnet wurden

$P_{aus} = \{p \in pz(m1) | pn_p \text{ hat Präfix „M1\_“}\} \backslash \{(M1\_request, 1, e), (M1\_busLock, 1, e)\}$ und die Menge der Eingangssignale des Masters, die noch keiner Phase zugeordnet wurden

$P_{ein} = \{p \in pz(m1) | pn_p \text{ hat Präfix „OPB\_“}\} \backslash \{(OPB\_grant, 1, a)\}$ erfasst.

Zu Beginn der $xferM1$-Phase müssen all diese Ausgangssignale des Masters, und die Signale des Slaves, die das Ende anzeigen könnten gesetzt bzw. berechnet werden:

$PP_{xferM1S} = \{(OPB\_xferAck, 1, a), (OPB\_timeout, 1, a), (OPB\_retry, 1, a)\} \cup P_{aus}$

Am Ende müssen alle Signale des Slaves gesetzt bzw. berechnet werden:

$PP_{xferM1E} = P_{ein}$

Keine anderen Signale beeinflussen den Start oder das Ende der Phase:

$PP_{xferM1O} = \emptyset$

Die Beobachtungsdauer ist $tb_{xferM1} = 2$ und damit ergeben sich die Start-, End- und Abbruchkriterien, wie folgt:

Die Phase startet, wenn $M1\_request$ von Null auf Eins wechselt oder wenn im vorangegangenen Takt die Phase endete:

$$S_{xferM1} = \left\{ (w_1, w_2) \in \left(\bigtimes_{p \in P_{aus}} SI_p \times \bigtimes_{p \in PP_{ein}} SI_p \right)^2 \;\middle|\; \begin{array}{l} \left[\begin{array}{l} proj_{(M1\_request,1,e)}(w_1) = 0 \\ \vee \\ proj_{(OPB\_xferAck,1,a)}(w_1) = 1 \\ \vee \\ proj_{(OPB\_timeout,1,a)}(w_1) = 1 \\ \vee \\ proj_{(OPB\_retry,1,a)}(w_1) = 1 \end{array} \right] \\ \wedge \\ proj_{(M1\_request,1,e)}(w_2) = 1 \end{array} \right\}$$

Die Phase endet, wenn $M1\_request$ auf Eins ist und wenn dazu ein Signal des Slaves bzw. Busses das Ende markiert:

$$E_{xferM1} = \left\{ (w_1, w_2) \in \left(\bigtimes_{p \in P_{aus}} SI_p \times \bigtimes_{p \in PP_{ein}} SI_p \right)^2 \;\middle|\; \begin{array}{l} \left[\begin{array}{l} proj_{(OPB\_xferAck,1,a)}(w_2) = 1 \\ \vee \\ proj_{(OPB\_timeout,1,a)}(w_2) = 1 \\ \vee \\ proj_{(OPB\_retry,1,a)}(w_2) = 1 \end{array} \right] \\ \wedge \\ proj_{(M1\_request,1,e)}(w_2) = 1 \end{array} \right\}$$

Die Phase bricht ab, wenn $M1\_request$ von Eins auf Null wechselt und die Phase im vorangegangenen Takt nicht endete.

$$A_{xferM1} = \left\{ \begin{array}{l|l} (w_1, w_2) \in & proj_{(OPB\_xferAck,1,a)}(w_1) = 0 \\ \left(\bigtimes_{p \in P_{aus}} SI_p \times \bigtimes_{p \in PP_{ein}} SI_p \right)^2 & \wedge \\ & proj_{(OPB\_timeout,1,a)}(w_1) = 0 \\ & \wedge \\ & proj_{(OPB\_retry,1,a)}(w_1) = 0 \\ & \wedge \\ & proj_{(M1\_request,1,e)}(w_1) = 1 \\ & \wedge proj_{(M1\_request,1,e)}(w_2) = 0 \end{array} \right\}$$

Für alle Ports der Phase wähle ich Null als den Defaultzustand, da so die Phase nicht endet und nicht startet. Für alle weiteren Master am OPB ergeben sich die Phasen analog.

D.4. Phasen im OPB-Slave-Interface

Das Slaveinterface des OPB unterscheidet sich vom Masterinterface lediglich dadurch, dass das $Mn\_request$- und das $OPB\_MnGrant$-Signal entfallen, da ein Slave nicht den Zugriff auf den Bus anfordern bzw. erhalten kann. Ausserdem werden in allen Signale mit $Mn\_$-Präfix das $Mn\_$-Präfix durch ein $OPB\_$-Präfix ersetzt und die Richtung invertiert, da diese Signale vom OPB zum Slave verlaufen. Entsprechend werden in allen Signalen des Masterinterfaces mit $OPB\_$-Präfix das $OPB\_$-Präfix durch ein $Sn\_$-Präfix ersetzt und die Richtung invertiert, da diese Signale von Slave n getrieben werden. Definiert man für ein das Slaveinterface von Slave $s1$ den Busphasensatz, so erhält man im Grunde den gleichen Satz wie für den Master, aber ohne die Phasen für $request$ und $grant$, also nur $lockS1$ und $xferS1$.

D.5. Effekt der J-R-Simulation des OPB

Nehmen wir an, es gibt ein taktgenaues Simulationsmodell A des OPB und ein taktgenaues J-R-Simulationsmodell B des OPB zur Performance-Evaluation, für das J und R mit Hilfe der oben beschriebenen Busphasen (für alle Master und Slaves) definiert sind. Abbildung D.1 zeigt ein Beispiel für einen Ablauf einer OPB-Kommunikation am Interface von Master $m1$, so wie er von Modell A berechnet wird. Die Inputs haben das Präfix M1_, die Outputs das Präfix OPB_.

Die gleiche Kommunikation soll mit Hilfe von Modell B simuliert werden. Die Takte, in denen Erfüllungen der Phasenstart und -endkriterien vorliegen, sind mit grünen bzw. roten Ellipsen markiert. Dabei umschließen die Ellipsen stest die Ports, deren Werte in diesem Takt explizit gesetzt bzw. berechnet werden müssen. Man erkennt, dass die Inputabläufe nicht der J-Einschrämkung genügen, da sich M1_ABus und M1_DBus vor dem Start der Phase $xfer$ ändern und da sich nach dem Ende und vor dem erneuten Start der Phase $xfer$

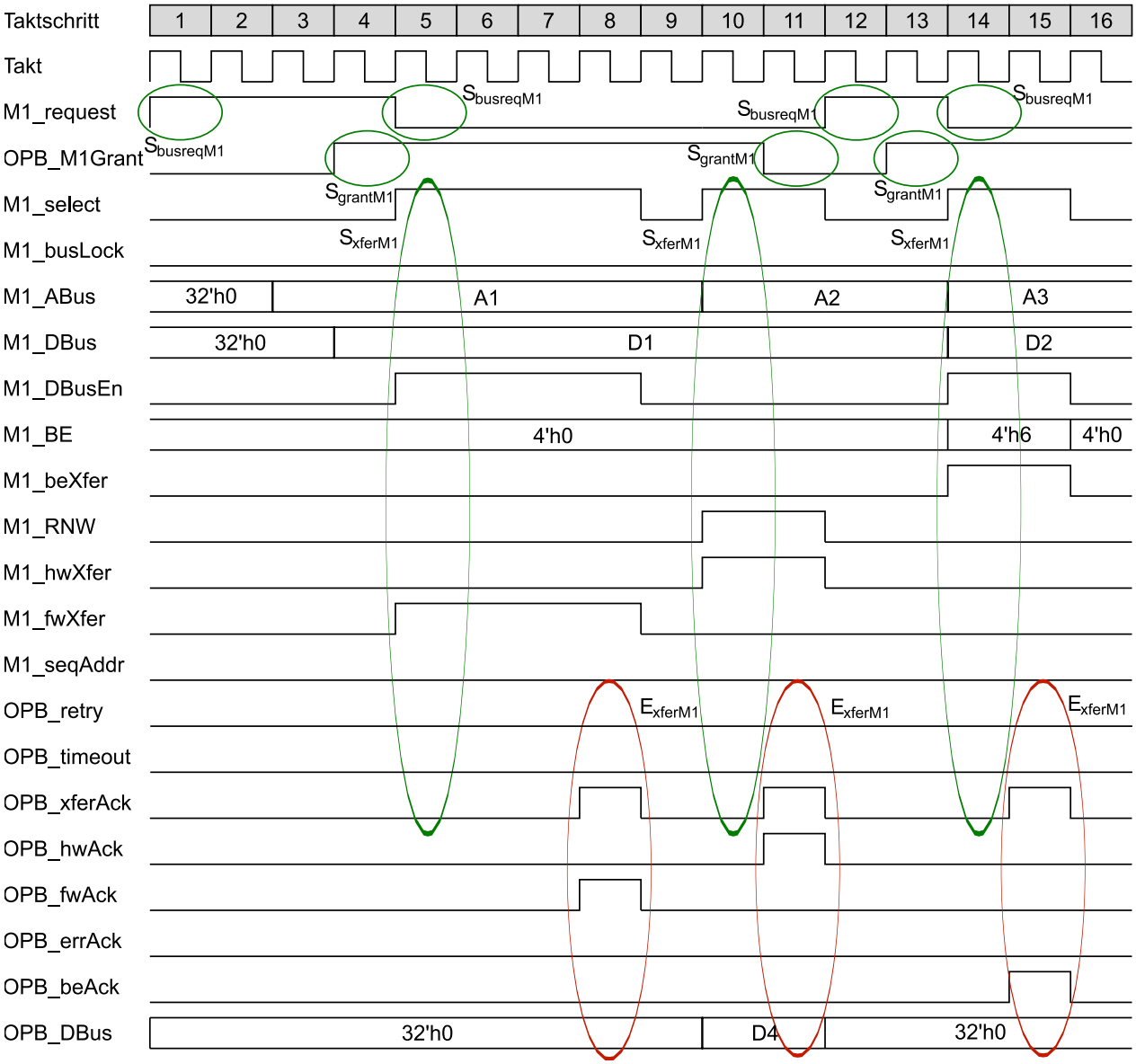


Abbildung D.1.: Full-Word-Write, Half-Word-Read und Byte-Enabled-Write auf dem OPB

einige Signale auf den Default-Wert ändern, während andere unverändert bleiben, also nicht der volle Default-Zustand der Phase $xfer$ eingenommen wird. Damit der Kommunikationsautomat des OPB durch eine J-R-Simulation simuliert werden kann, muss eine Filterung wie in Abschnitt 3.2.1 beschrieben, durchgeführt werden.

Abbildung D.2 zeigt den Ablauf der Kommunikation, den Modell A mit vorgeschalteten Filtern liefert. Man erkennt nun, dass der Input der J-Einschränkung genügt. Man erkennt auch eine Änderung der Outputs, die darauf zurückzuführen ist, dass auch die Inputs der Slaves gefiltert werden und sich dort die Daten nur am Phasenende ändern dürfen. In Hinsicht auf die Transferdauern und auch die übermittelten Daten hat die Filterung aber keine Auswirkung. Dies war zu erwarten, da die Änderungen durch die Filter nur in „don't-care"-Zuständen erfolgen. Es sei darauf hingewiesen, dass Performance-Modelle diese Filterung derart vornehmen können, dass sie nicht J-kompatible Änderungen schlicht ignorieren, in-

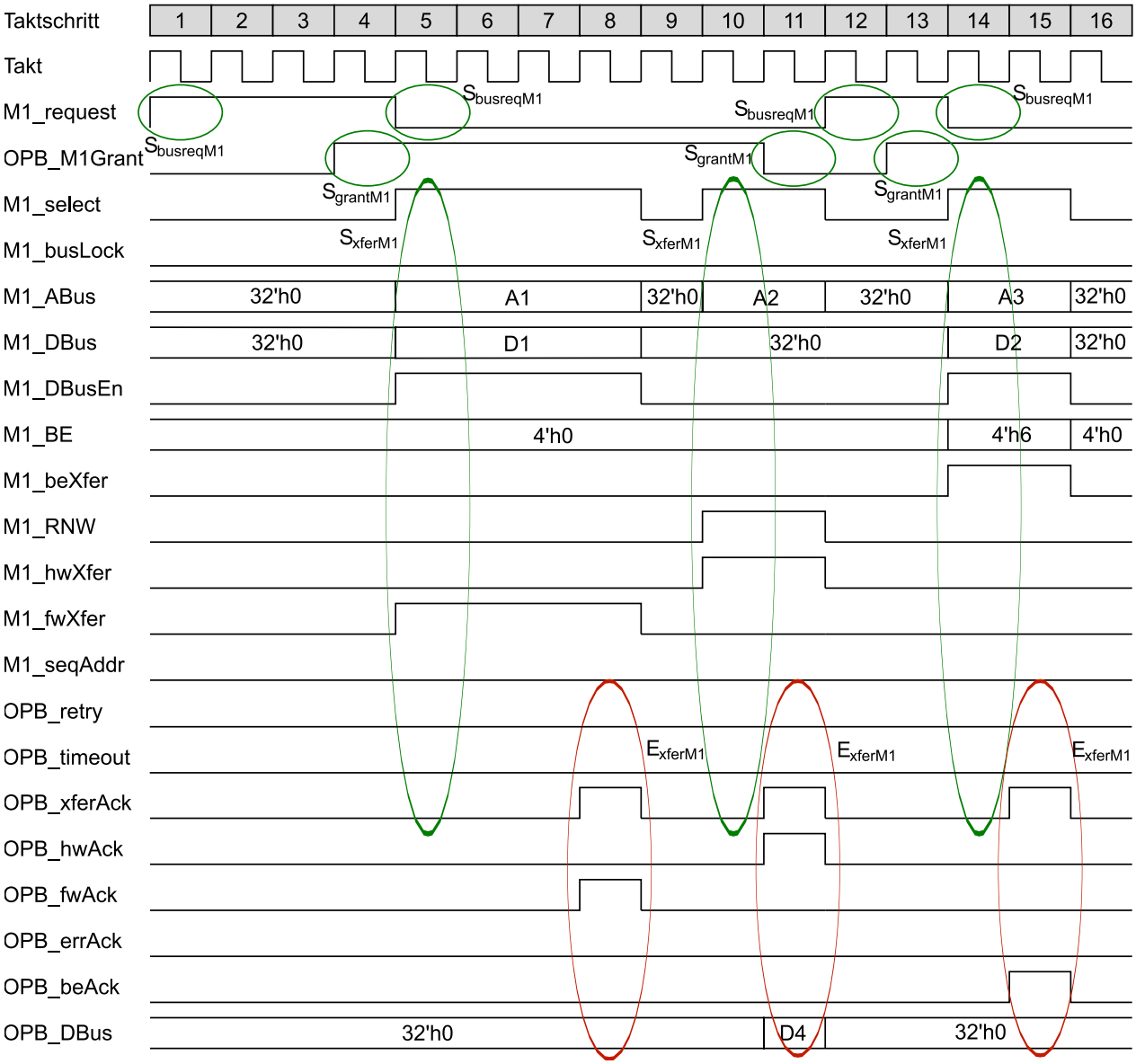


Abbildung D.2.: Full-Word-Write, Half-Word-Read und Byte-Enabled-Write auf dem OPB mit Inputfilter

dem sie nur zu Phasenstart- und -endzeitpunkten die Eingänge abfragen und zu allen anderen Zeitpunkten J-Kompatibilität annehmen. Dann ist die Filterung nur noch rein konzeptuell vorhanden und führt zu keinerlei zusätzlichem Simulationsaufwand.

Dem J-R-Modell B werden dann nur zu den in Tabelle D.3 gelisteten Takten Inputs übergeben. Dabei listet die Tabelle auch den Umfang der übergebenen Information.

Man erkennt, dass dem J-R-Modell B in lediglich 5 Takten Inputs übergeben werden müssen. Die Signaländerungen in die Default-Zustände kann das J-R-Modell aus den vom ihm selbst errechneten Outputs ableiten und braucht diese nicht explizit übergeben zu bekommen. Bei der taktgenauen Simulation ohne Input-Filter müssen 9 Inputs übergeben werden (jeder Takt in dem eine Inputänderung in Abbildung D.1 auftritt) und bei der taktgenauen Simulation mit Inputfilter sind es 7 (jeder Takt in dem eine Inputänderung in Abbildung D.2 auftritt).

Taktschritt	Input	Bedeutung
1	M1_request=1	S_{busreq}
5	M1_request=0	E_{busreq}
	M1_select=1, M1_busLock=0, M1_ABus=A1, M1_DBus=D1, M1_DBusEn=1, M1_BE=0, M1_beXfer=0, M1_RNW=0, M1_hwXfer=0, M1_fwXfer=1, M1_seqAddr=0	S_{xfer}
10	M1_select=1, M1_busLock=0, M1_ABus=A2, M1_DBus=0, M1_DBusEn=0, M1_BE=0, M1_beXfer=0, M1_RNW=1, M1_hwXfer=1, M1_fwXfer=0, M1_seqAddr=0	S_{xfer}
12	M1_request=1	S_{busreq}
14	M1_request=0	E_{busreq}
	M1_select=1, M1_busLock=0, M1_ABus=A3, M1_DBus=D2, M1_DBusEn=1, M1_BE=6, M1_beXfer=1, M1_RNW=0, M1_hwXfer=0, M1_fwXfer=0, M1_seqAddr=0	S_{xfer}

Tabelle D.3.: Komprimierter Inputablauf

Tabelle D.4 zeigt die von Modell B zu erwartende Ausgabe. Man erkennt, dass für jeden nach R bedeutsamen Taktschritt, also für alle Taktschritte in denen Phasen starten, abbrechen oder enden, die Outputs für die entsprechenden Phasen berechnet werden. Es werden insgesamt für 8 Taktschritte Outputs generiert. Geht man davon aus, dass auch in jedem Takt in dem Inputs übergeben werden Berechnungen stattfinden, wird das J-R-Simulationsmodell also in 10 Takten Berechnungen durchführen. Bei der taktgenauen Simulation ohne Inputfilter (Abbildung D.1) müssen mindestens 13 mal Berechnungen durchgeführt werden (bei jeder Inputänderung und bei Outputänderungen ohne gleichzeitige Inputänderung)[2], während bei der taktgenauen Simulation mit Inputfilter (Abbildung D.2) mindestens 12 mal Outputs berechnet werden müssen[2]. Darüber hinaus berechnet die taktgenaue Simulation grundsätzlich alle Ausgabeports, während die J-R-Simulation nur eine Teilmenge der Ausgabeports berechnet.

Da das J-R-Modell nicht zu allen Takten Inputs bekommt und Outputs erzeugt, können dessen Inputs und Outputs nur in abstrakter Form (z.B. die vorliegenden Tabellen) dargestellt werden. Um wieder zu vollständigen Signalverläufen zu gelangen, müssen Input und Output der J-R-Simulation mit Hilfe der Default-Zustände, wie in Abschnitt 3.2.1 beschrie-

[2]Diese Anzahl gilt unter der Annahme der OPB arbeitet voll kombinatorisch. Ist eine einzige getaktete Ausgabe im OPB vorhanden, sind es 16 Berechnungen.

Taktschritt	Output	Bedeutung
4	OPB_grant=1	S_{grant}
5	OPB_xferAck=0, OPB_retry=0, OPB_timeout=0	S_{xfer}
8	OPB_xferAck=1, OPB_hwAck=0, OPB_fwAck=1, OPB_retry=0, OPB_timeout=0, OPB_errAck=0, OPB_beAck=0, OPB_DBus=0	E_{xfer}
10	OPB_xferAck=0, OPB_retry=0, OPB_timeout=0	S_{xfer}
11	OPB_grant=0	E_{grant}
11	OPB_xferAck=1, OPB_hwAck=1, OPB_fwAck=0, OPB_retry=0, OPB_timeout=0, OPB_errAck=0, OPB_beAck=0, OPB_DBus=D4	E_{xfer}
13	OPB_grant=1	S_{grant}
14	OPB_xferAck=0, OPB_retry=0, OPB_timeout=0	S_{xfer}
15	OPB_xferAck=1, OPB_hwAck=0, OPB_fwAck=0, OPB_retry=0, OPB_timeout=0, OPB_errAck=0, OPB_beAck=1, OPB_DBus=0	E_{xfer}

Tabelle D.4.: Output der *J-R*-Simulation

ben, dekomprimiert werden. Dies führt dann zur gleichen Darstellung wie in in Abbildung D.2.

Es wird deutlich, dass eine *J-R*-Simulation zur Performance-Evaluation mit einer geringeren Anzahl an Inputänderungen eine geringere Anzahl an Outputs berechnen muss und dabei nach wie vor die für die Performance relevanten Informationen zu liefern im Stande ist. Dies resultiert daraus, dass durch die Einschränkung der Inputs und Outputs auf Phasenstarts und -enden und durch die Definition von Default-Zuständen von vielen expliziten Signalzuweisungen abstrahiert werden kann. Dazu gehören auch bzw. gerade die Signalwechsel auf dem Taktsignal in Takten, in denen keine *J*-konformen Inputänderungen oder nach *R* relevanten Outputänderungen auftreten.

D.6. Power-Analyse

Zur Power-Analyse müssen für jeden Master und Slave die Eingabe- und Ausgabepowerphase definiert werden. Wie in Abschnitt 3.2.2 beschrieben, müssen dazu $PP_{pow_E M1}$ und $PP_{pow_A M1}$ festgelegt werden.

Angenommen die Datensignale M1_DBus und OPB_DBus sind signifikant für die Power-Analyse, da Änderungen dieser Signale zu powerintensiven Schaltvorgängen in den Datenmultiplexerns führen, dann kann man $PP_{pow_E M1} = \{(M1\_DBus, 32, e), (M1\_DBusEn, 1, e)\}$ und $PP_{pow_A M1} = \{(OPB\_DBus, 32, a)\}$ festlegen. Nimmt man die dadurch festgelegten

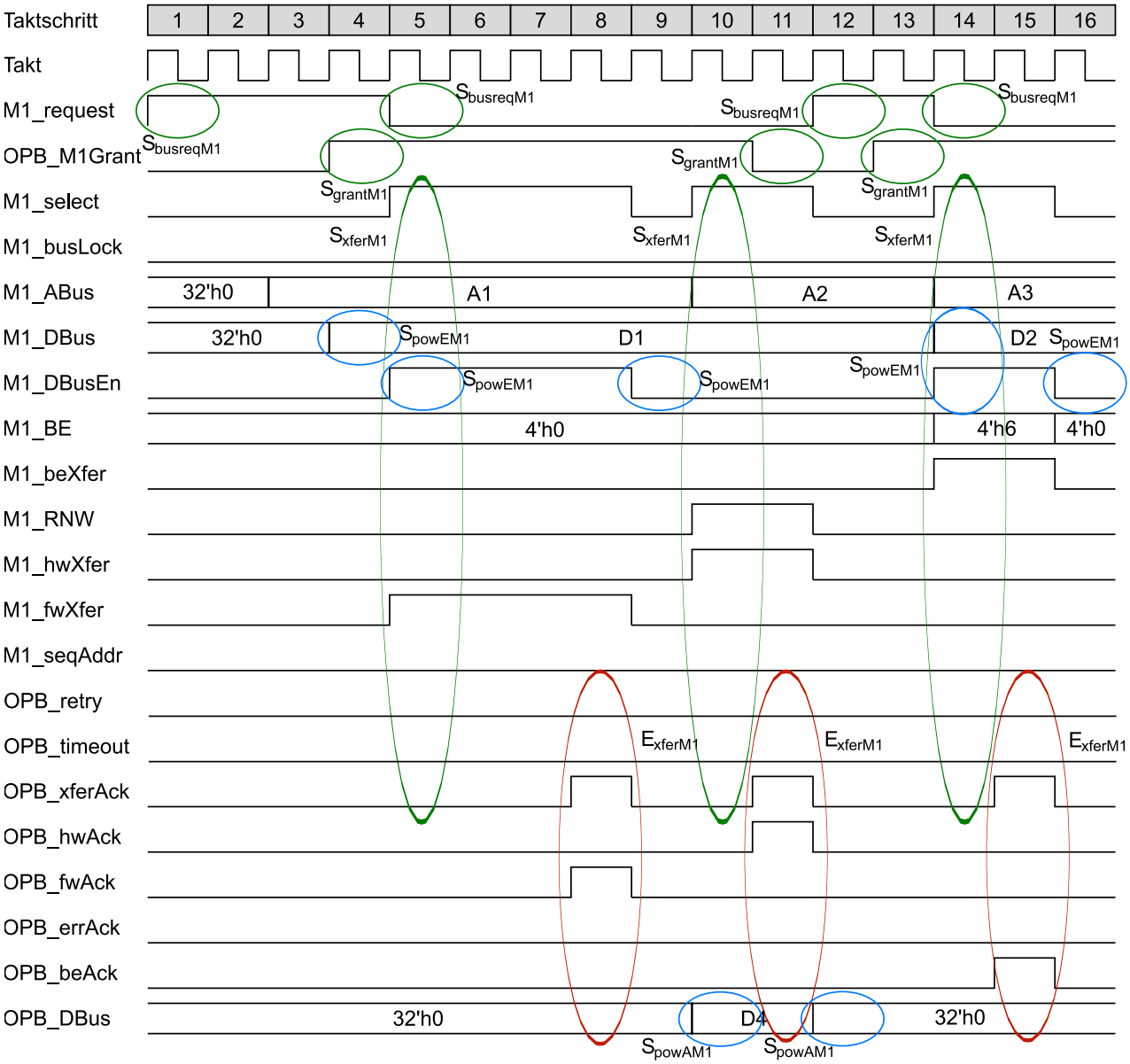


Abbildung D.5.: Full-Word-Write, Half-Word-Read und Byte-Enabled-Write auf dem OPB mit Powerphasen

Eingabe- und Ausgabepowerphasen in die Menge der Busphasen auf, so gibt es neue Phasenstarts, wie in Abbildung D.5 dargestellt (vgl. Abbildung D.1).

Eine Inputfilterung zur J-Kompatibilität würde nunmehr nur noch die Änderung auf dem Addressbus in Takt 3 unterdrücken. Dementsprechend müssen an das Powermodell nun 8 Inputs übergeben werden, da die Takte 4, 9 und 16 aufgrund der Eingabe-Powerphase bedeutsam sind. Darüber hinaus muss die J-R-Simulation nun 11 Outputs berechnen, da die Takte 12 und 16 aufgrund der Ausgabe-Powerphase bedeutsam sind. Dies ist nur noch geringfügig weniger als für die taktgenaue Simulation (9 Inputs und 13 Outputs). Es wird deutlich, dass die mögliche Abstraktion für die Power-Analyse, verglichen mit der der Performance-Analyse, geringer ist.

E. Erweiterungen des Generic Payload: Details

Inhalt

E.1. Einleitung

In Abschnitt 2.3.2 wurde das Konzept des Erweiterungsmechanismus des Generic Payload eingeführt. Dieser Anhang erläutert einige technische Details des Mechanismus und hebt die sich daraus ergebenden Gefahren für die korrekte und stabile Simulation hervor.

E.2. Grundlegendes Konzept

Wie bereits in Abschnitt 2.3.2 erläutert, ist eine GP-Erweiterung eine Instanz einer von `tlm_extension` abgeleiteten Klasse. Abbildung E.1 zeigt dies für eine GP-Erweiterung mit dem Klassennamen `my_extension`.

Die abstrakte Basisklasse `tlm_extension_base` enthält drei virtuelle Funktionen, die vom Designer einer Erweiterung überschrieben werden müssen bzw. können:

`tlm_extension_base* clone()` : Diese Funktion muss überschrieben werden. Ein Aufruf von `clone` soll grundsätzlich eine Kopie der Erweiterung zurück liefern, die dann in einer Kopie des GP, das die Originalerweiterung enthielt, verwendet werden kann. Sie muss derart erzeugt werden, dass ein Aufruf von `free` (siehe unten) diese korrekt wieder löscht bzw. frei gibt. Die Funktion darf `NULL` zurück liefern, wenn eine Kopie eines GP, das die Erweiterung enthielt, keine Instanz der Erweiterung enthalten soll.

`void copy_from(tlm_extension_base&)` : Diese Funktion muss überschrieben werden. Ein Aufruf von `copy_from` soll den Inhalt der Erweiterung, auf der `copy_from` aufgerufen wurde, auf den Inhalt der Erweiterung, die als Argument übergeben wird, setzen. Diese Funktion wird verwendet, wenn ein GP Änderungen übernehmen soll, die einer Kopie dieses GP nach dem Kopiervorgang widerfahren sind.

`void free()` : Diese Funktion kann überschrieben werden. Ein Aufruf von `free` soll die Erweiterung entsprechend ihrer Speicherverwaltung wieder freigeben. Dies muss die inverse Operation zur vorgesehenen Art der Erzeugung sein. Die vorgesehene Art der Erzeugung wird einerseits von Initiatoren direkt und andererseits innerhalb von `clone` verwendet. Wird die Funktion nicht überschrieben, führt `free` eine `delete` aus. Das bedeutet, dass eine solche Erweiterung immer mit `new` erzeugt werden muss.

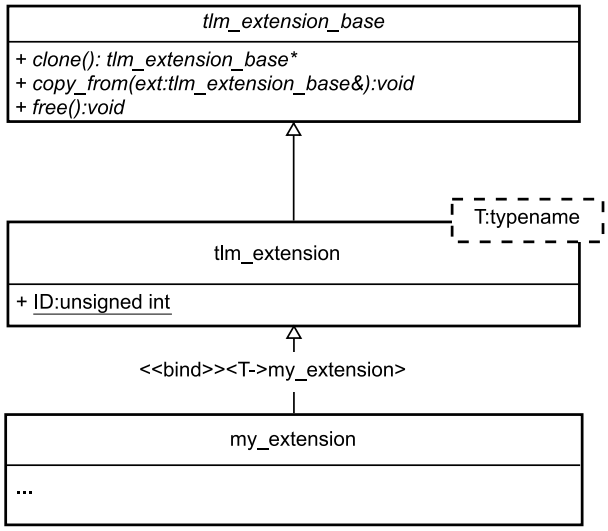


Abbildung E.1.: Klassendiagramm einer GP-Erweiterung

Man erkennt in Abbildung E.1, dass es sich bei `tlm_extension` um eine Template-Klasse handelt, die ein Klassenattribut `ID` enthält. Leitet eine Klasse von `tlm_extension` ab, um eine korrekte GP-Erweiterung zu werden, so legt sie den Template-Parameter auf ihren eigenen Typ fest (siehe Abbildung E.1). Dadurch existiert für jede Ableitung von `tlm_extension` eine unabhängige Ausprägung der Template-Klasse `tlm_extension` und somit auch ein unabhängiges Klassenattribut `ID` für jede dieser Ausprägungen. Die statische Initialisierung für diese Klassenattribute erfolgt derart, dass allen ein eindeutiger Wert zugewiesen wird und dass all diese Werte in einem von Null beginnenden zusammenhängenden Intervall liegen. Die Obergrenze dieses Intervalls wird durch die Gesamtzahl der Klassen, die wie in Abbildung E.1 dargestellt von `tlm_extension` ableiten, festgelegt. Der exakte Wert des Klassenattributes ist nicht vorhersagbar, denn er hängt wie erwähnt von der Gesamtzahl vorhandener GP-Erweiterungsklassen und der Compiler-abhängigen Ausführungsreihenfolge des statischen Initialisierungscodes ab. Listing E.2 verdeutlicht dies.

Die Tatsache, dass alle GP-Erweiterungen von der Template-freien Klasse `tlm_extension_base` abgeleitet sind, erlaubt es, ein Feld von Zeigern auf Objekte des Typs `tlm_extension_base` anzulegen. Die eindeutige ID einer jeden GP-Erweiterungsklasse kann dann als Index dieser Klasse im Feld verwendet werden. Dies wird in Listing E.3 verdeutlicht. Man erkennt, wie GP-Erweiterungen sehr einfach in das Feld eingetragen werden. Das Auslesen ist etwas komplexer, da der Zeiger aus dem Feld erst noch in den korrekten Typ gecastet

werden muss. Man erkennt auch, dass beim Eintragen die ID entweder über die Instanz oder über die Klasse abgerufen werden kann, während dies beim Auslesen nur über die Klasse geht, da vor dem Lesen noch keine Instanz existiert, die den ID-Zugriff erlauben würde. Jedes GP enthält ein mit dem in Listing E.3 vergleichbares Feld von GP-Erweiterungen.

```
1  //Definition zweier GP-Erweiterungsklassen entsprechend TLM-2.0-LRM
2  // (vgl. Abbildung E.1)
3  struct my_extension : public tlm_extension<my_extension>{/*Implementierung*/};
4  struct my_other_extension : public tlm_extension<my_other_extension>{/*Implementierung*/};
5
6  my_extension ext1; //eine Instanz von my_extension
7  my_extension ext2; //eine weitere Instanz von my_extension
8  my_other_extension ext3; //eine Instanz von my_other_extension
9
10 //Zugriff auf die eindeutige ID ueber den Klassennamen
11 unsigned int my_extension_ID=my_extension::ID;
12 unsigned int my_other_extension_ID=my_other_extension::ID;
13
14 //Die IDs sind immer verschieden
15 if (my_extension_ID==my_other_extension_ID) std::cout<<"Das kann nie passieren!!"<<std::endl;
16
17 //Zugriff auf die ID ueber Instanzen
18 //Die IDs von Instanzen entsprechen stest der Klassen-ID
19 if (ext1.ID!=my_extension_ID) std::cout<<"Das kann nie passieren!!"<<std::endl;
20 if (ext2.ID!=my_extension_ID) std::cout<<"Das kann nie passieren!!"<<std::endl;
21 if (ext3.ID!=my_other_extension_ID) std::cout<<"Das kann nie passieren!!"<<std::endl;
22 if (ext1.ID!=ext2.ID) std::cout<<"Das kann nie passieren!!"<<std::endl;
23
24 //Instanzen unterschiedlicher GP-Erweiterungen haben immer unterschiedliche IDs
25 if (ext3.ID==ext1.ID) std::cout<<"Das kann nie passieren!!"<<std::endl;
```

Listing E.2: Beispiel zum ID-Attribut von GP-Erweiterungen

```
1  //Definition zweier GP-Erweiterungsklassen
2  struct my_extension : public tlm_extension<my_extension>{/*Implementierung*/};
3  struct my_other_extension : public tlm_extension<my_other_extension>{/*Implementierung*/};
4
5  my_extension ext1; //eine Instanz von my_extension
6  my_other_extension ext2; //eine Instanz von my_other_extension
7
8  //ein GP-Erweiterungsfeld fuer bis zu 100 verschiedene Erweiterungen
9  typedef tlm_extension_base* tlm_extension_base_ptr;
10 tlm_extension_base_ptr* extension_array=new tlm_extension_base_ptr[100];
11
12 //fuege ext1 in die Feldposition ein, die fuer die GP-Erweiterung von Typ my_extension vorgesehen ist
13 extension_array[ext1.ID]=&ext1;
14
15 //fuege ext2 in die Feldposition ein, die fuer die GP-Erweiterung von Typ my_other_extension vorgesehen ist
16 extension_array[my_other_extension::ID]=&ext2;
17
18 //lies ext1 wieder aus dem Feld aus
19 my_extension* ext1_ptr=static_cast<my_extension*>(extension_array[my_extension::ID]);
```

Listing E.3: Beispiel für ein Feld von GP-Erweiterungen

E.3. Shared-Data-Problematik

Der vorangegangene Abschnitt hat das zugrundeliegende Konzept für GP-Erweiterungen illustriert. Das Potential des Mechanismus ist immens. Es gibt keine Einschränkung bezüglich der als GP-Erweiterungen definierbaren Information. Dies können einfache Daten, wie Integer Datentypen oder Boolsche Variablen sein oder komplexe Klassen mit vielen Attributen und Funktionen. Im Grunde kann das GP mit diesem Mechanismus für jeden beliebigen Einsatz, sogar weit über die Modellierung von MMB hinaus eingesetzt werden, da eine weder in Typ noch Anzahl begrenzte Menge an Elementen hinzugefügt werden kann. Jedoch birgt der Mechanismus auch Gefahren.

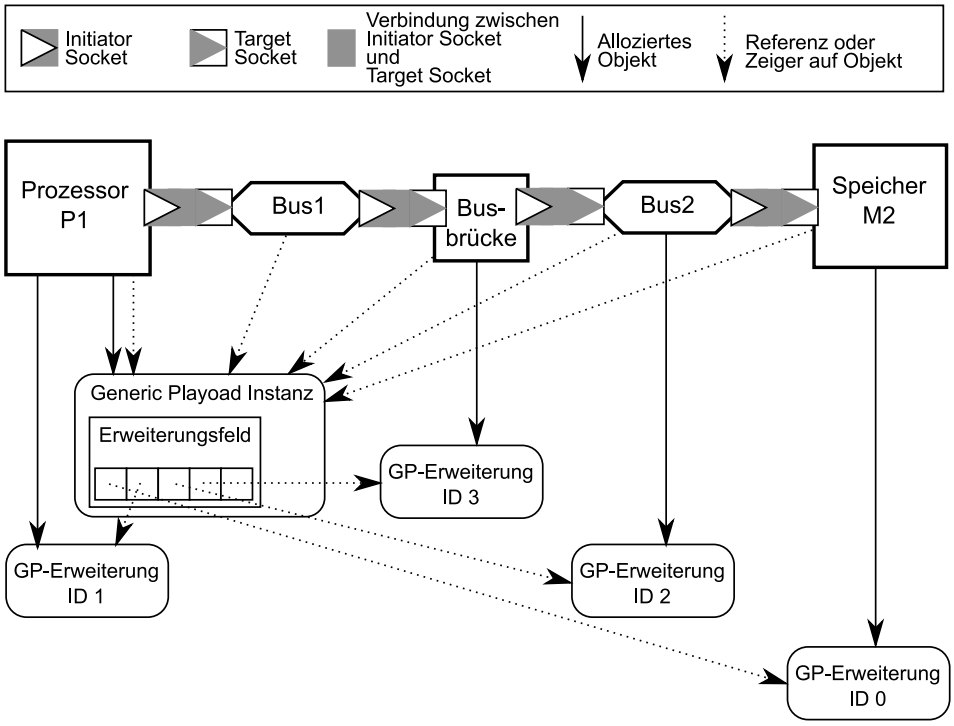


Abbildung E.4.: Shared-Data bei einer TLM-2.0-Transaktion

Diese sollen Anhand eines Beispiels illustriert werden. Man betrachte Abbildung E.4. Darin ist eine Situation skizziert, wie sie sich bei einer TLM-2.0-Transaktion ergeben kann: Das Generic Payload ist bereits einmal vom Prozessor P1 zum Speicher M2 übertragen worden, z.B. während der Phase `BEGIN_REQ`. Nun haben alle Module auf dem Transaktionspfad eine Referenz auf das Generic Payload. Zusätzlich haben die Module P1, Busbrücke, Bus2 und M2 GP-Erweiterungen hinzugefügt. P1 kann z.B. eine Priorität gesetzt haben, währende die Busbrücke ein Bus-Lock an Bus2 signalisieren möchte usw.

Man erkennt, dass alle Module über das Generic Payload auch einen Zeiger auf die GP-Erweiterungen erhalten können (da das Erweiterungsfeld, wie im vorangegangenen Abschnitt beschrieben aus Zeigern besteht). Dementsprechend muss nicht nur das Generic Payload, sondern auch alle GP-Erweiterungen als sog. „Shared-Data“ betrachtet werden. D.h. es gibt ein

Modul, das ein Erweiterungsobjekt alloziert und dessen Existenz kontrolliert, während alle anderen Module sich dann dieses Objekt teilen. In den verschiedenen Modulen existieren parallele Simulationsprozesse, sodass sich hier die Problematik von Shared-Data bei parallelen Prozessen ergibt: Die Ausführungsreihenfolge der Prozesse ist im Allgemeinen nicht bekannt und so kann eine Änderung eines Wertes einer GP-Erweiterung bedeuten, dass ein anderer Prozess einen Wert „übersieht“, da er bisher noch gar nicht ausgeführt wurde. Ein solches Überschreiben und Übersehen von Werten kann zu fehlerhaft simuliertem Verhalten führen und ist extrem schwer zu debuggen, da das Überschreiben weit vor (ggf. mehrere simulierte Zeitschritte) dem Bemerken des Fehlers liegt.

Die schlimmst-mögliche Entartung eines falschen Zugriffes auf Shared-Data entsteht, wenn das Objekt de-alloziert wird und dann erst ein Zugriff erfolgt. Dies führt zu völlig unvorhersagbarem Verhalten (Verarbeitung von Daten aus nicht-alloziertem Speicher) oder zu Simulatorabstürzen. Auch diese Fehler sind sehr schwer aufzufinden. Darüber hinaus ergibt sich die Problematik, dass, wie im vorherigen Abschnitt deutlich wurde, immer nur ein Zeiger pro Erweiterungsklasse im Feld existieren kann. Ein Ersetzen eines vorhandenen Zeigers kann zu Speicherlecks oder Speicherverunreinigung führen (abhängig vom Memory-Management der Erweiterung). Folglich müssen also zwei Module die gleiche Shared-Data-Erweiterung nutzen, wenn sie die gleiche Information (z.B. einen speziellen Error-Code) übermitteln wollen. Sie haben keine Möglichkeit jeweils eine eigene Instanz zu nutzen.

Diese Probleme wiegen bei der taktgenauen Simulation besonders schwer, da die Transaktion gleichzeitig auf mehreren Punkt-zu-Punkt-Verbindungen (in Abbildung E.4 sind dies 4: P1↔Bus1, Bus1↔Busbrück, Busbrücke↔Bus2 und Bus2↔M2) potentiell in verschiedenen Phasen aktiv ist. Sie kann also z.B. zwischen P1 und Bus1 bereits beendet sein, während sie noch zwischen Bus2 und M2 läuft. Es ist also wichtig, dass in diesem Fall P1 die Lebensdauer seiner Erweiterung an die Lebensdauer der Transaktion koppelt um Fehlzugriffe zu vermeiden.

Wird ein Modell von einem einzigen Ingenieur erstellt (das schließt die Definition aller Erweiterungen mit ein) ist er oder sie natürlich in der Lage für sich selbst Lösungen zu den gegebenen Problemen zu finden. Bei TLM steht aber auch der Austausch von IP-Blöcken im Vordergrund, das heißt, verschiedene Entwickler sind beteiligt. In der Regel definiert ein Ingenieur(-Team) die TLM-Modellierung des Protokolls (also auch die Erweiterungen), während andere Ingenieure IP-Blöcke für dieses Protokoll entwickeln. In diesem Fall sind unmissverständliche Regelungen zur Vermeidung der beschriebenen Probleme zwingend. Solche Regeln sollten nicht für jedes Protokoll neu definiert werden, da dann das Erlernen und Verwenden verschiedener Protokolle erschwert wird.

Zusammenfassend kann man also sagen, dass zur sicheren Verwendung von GP-Erweiterungen klare Regeln existieren müssen, die festlegen, unter welchen Umständen Inhalte von Erweiterungen sicher gelesen und sicher verändert werden können und wie die im GP vorhandenen Mechanismen zum Memory-Management eingesetzt werden können. Diese Regeln müssen an eine im Rahmen der taktgenauen busphasenbasierten Modellierung allgemein-

gültige Klassifikation von Erweiterungen angepasst sein. Somit kann ein Entwickler, wenn er im Rahmen des GP-Mappings (siehe Abschnitt 4.3.1) eine Erweiterung identifiziert hat, diese klassifizieren und anschließend anhand der Regeln implementieren und entsprechende Zugriffsregeln übernehmen.

F. Code-Beispiel zur L1- und L2-Interoperabilitätsprüfung

Inhalt

F.1. Einleitung

In Abschnitt 4.6 wurden Bindungschecks zur Laufzeit eingeführt, die die Tests zur Kompilierzeit, so wie sie in TLM-2.0 existieren, erweitern. Dieser Abschnitt zeigt die Verwendung dieser Tests anhand eines kleinen Beispiels.

F.2. #include-Direktiven

Wie in Listing F.1 zu sehen, müssen lediglich die von mir entsprechend Abschnitt 4.6.5 überarbeiteten Sockets aus den TLM-Utilities inkludiert werden.

```
1 #include "simple_initiator_socket.h"
2 #include "simple_target_socket.h"
```

Listing F.1: Include-Direktiven für das L1-L2-Bindungstest-Beispiel

F.3. Zusätzliche Phasen und GP-Erweiterungen und eine entsprechende Traits-Class

Damit die L1-Bindungstests anschaulich demonstriert werden können, werden wie in Listing F.2 zu sehen, eine neue TLM-Phase und eine GP-Erweiterung definiert. Dazu wird noch eine TC definiert, die die Verwendung von GP und TLM-Phase sowie der in Abschnitt 4.6.5 definierten L1-Konfiguration bestimmt.

```
1  //Eine simple GP-Erweiterung
2  struct my_ext: public tlm::tlm_extension<my_ext>
3  {
4      virtual tlm_extension_base* clone() const {return NULL;}
5      virtual void copy_from(tlm_extension_base const &) {}
6      virtual ~my_ext(){}
7  };
8
9  //Eine zusaetlzliche Phase
10 DECLARE_EXTENDED_PHASE(ABORT_REQ);
11
12 //Eine TC, die die L1-Konfiguration verwendet
13 // Die Verwendung dieser TC debdeutet, dass alle Phasen des BP unverhandelbar sind
14 // und dass ABORT_REQ und my_ext verhandelbar sind.
15 struct my_protocol_types
16 {
17   typedef tlm::tlm_generic_payload tlm_payload_type;
18   typedef tlm::tlm_phase tlm_phase_type;
19   typedef catlm::L1_config tlm_rt_bind_config_type;
20 };
```

Listing F.2: GP-Erweiterung und zusätzliche Phase für das L1-L2-Bindungstest-Beispiel

F.4. L2-Basisklassen

In Listing F.3 sind zwei einfache Basisklassen für L2-Tests zu sehen. Man beachte, dass beide virtuell von `L2_base` ableiten, damit sicher gestellt ist, dass nur ein Objekt von `L2_base` existiert, wenn von beiden Klassen abgeleitet wird.

```
1  //Eine L2-Klasse zum Vergleich von Addressbreiten
2  class address_width_check : public virtual catlm::L2_base
3  {
4  public :
5    unsigned int get_address_width(){return m_address_width;}
6  protected:
7    unsigned int m_address_width;
8  };
9
10 //Eine L2-Klasse zum Vergleich einer anderen Signalbreite
11 class another_width_check : public virtual catlm::L2_base
12 {
13 public :
14   unsigned int get_another_width(){return m_another_width;}
15 protected:
16   unsigned int m_another_width;
17 };
```

Listing F.3: L2-Basisklassen für das L1-L2-Bindungstest-Beispiel

F.5. Initiator-Modul

Listing F.4 zeigt ein Initiator-Modul, das über die durch die Traits-Class bestimmten L1-Verhandlungen hinaus auch einen L2-Test bezüglich der Adressbreite durchführt. Da die Adresse im GP ein 64-Bit-Datentyp ist, wird im Zweifelsfalle eine Adressbreite von 64 Bit angenommen.

```
1  //Ein Initiator-Modul
2  SC_MODULE(init), public address_width_check //unterstuetzt L2-Tests bezueglich Addressbreiten
3  {
4    typedef tlm_utils::simple_initiator_socket<init,32,my_protocol_types> isock_type;
5    isock_type socket; //der Initiatorsocket
6    catlm::L1_config m_cfg; //die L1-Konfiguration fuer den Socket
7
8    SC_CTOR(init) : socket("socket")
9    {
10     SC_THREAD(run);
11
12     address_width_check::m_address_width=16; //setzen der modellierten Addressbreite
13
14     socket.register_nb_transport_bw(this, &init::nb_trans); //registrieren des transport-Callbacks
15     socket.register_get_config_bw(this, &init::get_config); //registrieren des get_config-Callbacks
16     socket.set_rt_bind_config_ptr(&m_cfg); //zuweisen der L1-Konfiguration an den Socket
17
18     m_cfg.set_extension_exigency(my_ext::ID, catlm::catlm_mandatory); //my_ext ist zwingend
19     m_cfg.set_phase_exigency(ABORT_REQ, catlm::catlm_mandatory); //ABORT_REQ ist zwingend
20
21     m_cfg.register_L2_callback(this, &init::L2_callback); //setzen des L2-Callbacks
22     m_cfg.set_L2_ptr(this); //setzen des L2-Objekts
23
24   }
25
26   //der Callback zur Rueckgabe der Konfiguration
27   const tlm::tlm_rt_bind_config_base* get_config(){return &m_cfg;}
28
29   //der L2-Callback
30   void L2_callback(catlm::L2_base* obj, unsigned int index){
31     address_width_check* addr_test=dynamic_cast<address_width_check*>(obj);
32
33     //Wenn die andere Seite keine explizite Addressbreite angibt, wird angenommen
34     //  dass die volle Breite des Datentyps uint64 unterstuetzt wird
35     unsigned int other_addr_width=addr_test?addr_test->get_address_width():64;
36     if (other_addr_width<address_width_check::m_address_width)
37     {
38       std::stringstream s;
39       s<<socket.name()
40        <<" address width ("<<address_width_check::m_address_width<<") is larger than address width of "
41        <<socket[index]->get_name_fw()<<" ("<<other_addr_width<<")";
42       SC_REPORT_WARNING("init",s.str().c_str());
43     }
44   }
45
46   void run(){...}
47
48   tlm::tlm_sync_enum nb_trans(tlm::tlm_generic_payload& gp, tlm::tlm_phase& ph, sc_core::sc_time& t){...}
49 };
```

Listing F.4: Initiator-Modul für das L1-L2-Bindungstest-Beispiel

F.6. Target-Modul

Listing F.5 zeigt ein Target-Modul, das über die durch die Traits-Class bestimmten L1-Verhandlungen hinaus auch einen L2-Test bezüglich der Adressbreite und einer weiteren, hier nicht näher bestimmten Breite durchführt. Da die Adresse im GP ein 64-Bit-Datentyp ist, wird im Zweifelsfalle eine Adressbreite von 64 Bit angenommen. Für die andere Breite wird im Zweifel 256 Bit angenommen.

Beim Start der Simulation testet das Target, wie der Master die Phase `ABORT_REQ` behandelt.

```
1  //Ein Target-Modul
2  SC_MODULE(target)
3    , public address_width_check //unterstuetzt L2-Tests bezueglich Addressbreiten
4    , public another_width_check //unterstuetzt L2-Tests bezueglich einer anderen Breite
5  {
6    typedef tlm_utils::simple_target_socket<target,32,my_protocol_types> tsock_type;
7    tsock_type socket; //der Targetsocket
8    catlm::L1_config m_cfg; //die L1-Konfiguration des Sockets
9
10   SC_CTOR(target) : socket("socket")
11   {
12     address_width_check::m_address_width=8;  //setzen der modellierten Addressbreite
13     another_width_check::m_another_width=128; //setzen der modellierten anderen Breite
14
15     socket.register_nb_transport_fw(this, &target::nb_trans); //registrieren des transport-Callbacks
16     socket.register_get_config_fw(this, &target::get_config); //registrieren des get_config-Callbacks
17     socket.set_rt_bind_config_ptr(&m_cfg);  //Zuweisen der L1-Konfiguration an den Socket
18
19     m_cfg.set_extension_exigency(my_ext::ID, catlm::catlm_mandatory); //my_ext ist zwingen
20     m_cfg.set_phase_exigency(ABORT_REQ, catlm::catlm_optional); //ABORT_REQ is optional
21
22     m_cfg.register_L2_callback(this, &target::L2_callback); //setzen des L2-Callbacks
23     m_cfg.set_L2_ptr(this); //setzen des L2-Objekts
24   }
25
26   //der Callback zur Rueckgabe der Konfiguration
27   const tlm::tlm_rt_bind_config_base* get_config()
28   {
29     return &m_cfg;
30   }
31
32   //der L2-Callback
33   void L2_callback(catlm::L2_base* obj, unsigned int index){
34     address_width_check* addr_test=dynamic_cast<address_width_check*>(obj);
35
36     //Wenn die andere Seite keine explizite Addressbreite angibt, wird angenommen
37     //  dass die volle Breite des Datentyps uint64 unterstuetzt wird
38     unsigned int other_addr_width=addr_test?addr_test->get_address_width():64;
39     if (other_addr_width>address_width_check::m_address_width)
40     {
41       std::stringstream s;
42       s<<socket[index]->get_name_bw()
43        <<" address width ("<<other_addr_width<<") is larger than address width of "
44        <<socket.name()<<" ("<<address_width_check::m_address_width<<")";
45       SC_REPORT_WARNING("target",s.str().c_str());
46     }
47
48     another_width_check* another_test=dynamic_cast<another_width_check*>(obj);
```

```
49
50    //Wenn die andere Seite keine explizite andere Breite angibt, wird angenommen
51    //  dass die die Breite von 256 Bit unterstuetzt wird
52    unsigned int other_width=another_test?another_test->get_another_width():256;
53    if (other_width>another_width_check::m_another_width)
54    {
55      std::stringstream s;
56      s<<socket[index]->get_name_bw()
57       <<" another width ("<<other_width<<") is larger than another width of "
58       <<socket.name()<<" ("<<another_width_check::m_another_width<<")";
59      SC_REPORT_WARNING("target",s.str().c_str());
60    }
61
62  }
63
64  void start_of_simulation()
65  {
66   for (unsigned int i=0; i<socket.size(); i++){
67     const catlm::L1_config* tmp=dynamic_cast<const catlm::L1_config*> (socket.get_resolved_rt_bind_config(i));
68     assert(tmp); //cast immer erfolgreich, da nur gebundene sockets abfragen werden und TC den Typ garantiert
69     switch (tmp->get_phase_exigency(ABORT_REQ)){
70     case catlm::catlm_mandatory:
71       std::cout<<socket[i]->get_name_bw()
72          <<" treats ABORT_REQ as mandatory. So I ("<<this->name()<<") will have to handle it."<<std::endl;
73       break;
74     case catlm::catlm_optional:
75       std::cout<<socket[i]->get_name_bw()
76          <<" treats ABORT_REQ as optional. So I ("<<this->name()<<") am not sure if it will be used or not."
77          <<std::endl;
78       break;
79     case catlm::catlm_rejected:
80       std::cout<<socket[i]->get_name_bw()
81          <<" treats ABORT_REQ as rejected. So I ("<<this->name()<<") will not need to handle it."<<std::endl;
82       break;
83     }
84   }
85  }
86
87  tlm::tlm_sync_enum nb_trans(tlm::tlm_generic_payload& gp, tlm::tlm_phase& ph, sc_core::sc_time& t){...}
88 };
```

Listing F.5: Target-Modul für das L1-L2-Bindungstest-Beispiel

F.7. sc_main und Simulationsausgabe

Listing F.6 zeigt die `sc_main`-Funktion zur Ausführung des Beispiels und Listing F.7 zeigt die resultierende Simulationsausgabe. Die L1-Tests sind erfolgreich, da sowohl Initiator als auch Target die Verwendung von `my_ext` als zwingend markiert haben. Bei der Verwendung von `ABORT_REQ` unterscheiden sich die Erfordernisgrade (Initiator: zwingend, Target: optional), jedoch ist dieser Unterschied akzeptabel und der resultierende Erfordernisgrad wird auf zwingend gesetzt. Das Target stellt beim Start der Simulation fest, dass der Master die Phase als zwingend betrachtet.

Vorher stellen sowohl Initiator als auch Target während der L2-Tests der Adressbreite fest, dass der Master einen größeren Adressraum bearbeitet als das Target und folglich Vorsicht

geboten ist. Außerdem stellt das Target fest, dass bei dem L2-Test von `another_width` (welcher vom Master nicht unterstützt wird und somit eine Breite von 256 angenommen wird) auch ein Problem vorliegt.

```
int sc_main (int, char **) {
  init i("i");
  target t("t");
  i.socket(t.socket);
  sc_core::sc_start();
  return 0;
}
```

Listing F.6: `sc_main` zur Verwendung von L1- und L2-Tests

```
            SystemC 2.2.0 --- Aug  5 2010 16:21:01
       Copyright (c) 1996-2006 by all Contributors
                    ALL RIGHTS RESERVED

Warning: init: i.socket address width (16) is larger than address width of t.socket (8)
In file: /Users/robertguenzel/mydev/cpp_systemc/tlm-2.0+ca/main.cpp:97

Warning: target: i.socket address width (16) is larger than address width of t.socket (8)
In file: /Users/robertguenzel/mydev/cpp_systemc/tlm-2.0+ca/main.cpp:168

Warning: target: i.socket another width (256) is larger than another width of t.socket (128)
In file: /Users/robertguenzel/mydev/cpp_systemc/tlm-2.0+ca/main.cpp:182

i.socket treats ABORT_REQ as mandatory. So I (t) will have to handle it.
```

Listing F.7: Simulationsausgabe des Beispiels zur Verwendung von L1- und L2-Tests

G. Experiment zur Erweiterungs-API-Performance

Inhalt

G.1. Einleitung

Das Ziel des Experiments ist die Untersuchung des Zugriffsoverheads, der durch die Erweiterungs-API aus Abschnitt 4.8.4 (Implementierung in [Günz10b]) entsteht. Dazu soll der absolute Overhead im Vergleich zu einer reinen Auto-Erweiterung vermessen und diskutiert werden. Zur Stützung der Diskussion werden zusätzlich die absoluten Kosten einer repräsentativen Aktivität innerhalb einer Kommunikation vermessen.

G.2. Aufbau des Experiments

Um den Vergleich fair zu gestalten, ist es wichtig, der reinen Auto-Erweiterung und den Zugriffen darauf die gleichen Fähigkeiten zu geben, wie der Erweiterungs-API aus Abschnitt 4.8.4. Damit die reine Auto-Erweiterung in ihrer Gültigkeit auch umschaltbar ist, muss sie einen `bool` Attribut enthalten, welches, wenn es auf `false` gesetzt ist, selbst bei Anwesenheit der Erweiterung Ungültigkeit anzeigt. Dies begründet sich dadurch, dass die Auto-Erweiterung, einmal hinzugefügt, nicht wieder manuell entfernt werden kann (siehe Abschnitt 4.8.4). Zur Erzeugung und Freigabe der Auto-Erweiterung wird der effiziente Objekt-Pool aus der `boost`-Bibliothek [Clea06]$_i$ benutzt.

Damit ergeben sich die Zugriffsfunktionen, wie sie in Listing G.1 enthalten sind. Daneben enthält Listing G.1 Kapselfunktionen für die entsprechenden Zugriffe mit Hilfe der Erweiterungs-API aus Abschnitt 4.8.4. Die Funktionen, die eine einzelne Auto-Erweiterung nutzen, tragen das Suffix `_pooled`, da die Auto-Erweiterung den `boost`-Pool nutzt. Die

Funktionen, die das von mir vorgeschlagene Konzept und die zugehörige API nutzen, tragen dementsprechend das Suffix `_api`. Die gezeigten Funktionen wurden in eine Bibliothek kompiliert, die dann vom eigentlichen Test benutzt wird. Da der Code in eine Bibliothek umgesetzt wird, kann der Compiler keine kontextspezifischen Optimierungen vornehmen. Würden die Funktionen dem Compiler direkt im Test zur Verfügung stehen, könnte er bei simplen Tests mittels Constant-Propagation oder Constant-Elimination Teile der Zugriffsfunktionen so stark optimieren, dass der Test verfälscht würde. Dies wird durch die Bibliothek verhindert und die Tests können sehr einfach gehalten werden.

```
1  struct test_extension_pool : public tlm::tlm_extension< test_extension_pool >
2  {
3  protected:
4     void copy_from(tlm::tlm_extension_base const & ext){
5      const test_extension_pool * t=static_cast<const test_extension_pool *>(&ext);
6      value=t->value; is_valid=t->is_valid;
7     }
8
9     tlm::tlm_extension_base* clone() const {
10      test_extension_pool * t=get_pool()->construct();
11      t->value=value; t->is_valid=is_valid;
12      return t;
13    }
14
15    void free(){ get_pool()->free(this); }
16
17    static extension_pool< test_extension_pool >* get_pool(){
18     static extension_pool< test_extension_pool > s_ptr(5);
19     return &s_ptr;
20    }
21 public:
22    static name * create_inst(){ return get_pool()->construct(); }
23    unsigned int value;
24    bool is_valid;
25 };
26
27 //Markiere Erweiterung als gueltig und liefere gleich als nutzbaren Zeiger zurueck
28 test_extension_pool* val_n_get_pooled(tlm::tlm_generic_payload& gp)
29 {
30   test_extension_pool* tmp;
31   gp.get_extension(tmp);
32   if (!tmp){
33     tmp=test_extension_pool::create_inst();
34     gp.set_auto_extension(tmp);
35   }
36   tmp->is_valid=true;
37   return tmp;
38 }
39
40 //Die gleiche Funktion wie oben, aber mit Hilfe der Erweiterungs-API
41 test_extension_pool* val_n_get_api(tlm::tlm_generic_payload& gp)
42 {
43   return catlm::extension_api::validate_and_get<test_extension_pool>(gp);
44 }
45
46 //Markiere eine Erweiterung eventuell erneut als gueltig
47 void revalidate_pooled(tlm::tlm_generic_payload& gp)
48 {
49   test_extension_pool* tmp;
```

```
50    gp.get_extension(tmp);
51    if (!tmp){
52      tmp=test_extension_pool::create_inst();
53      gp.set_auto_extension(tmp);
54    }
55    tmp->is_valid=true;
56  }
57
58  //Die gleiche Funktion wie oben, aber mit Hilfe der Erweiterungs-API
59  void revalidate_api(tlm::tlm_generic_payload& gp)
60  {
61    catlm::extension_api::validate<test_extension_pool>(gp);
62  }
63
64  //Rufe die Erweiterung aus dem GP ab
65  test_extension_pool* get_pooled(tlm::tlm_generic_payload& gp)
66  {
67    test_extension_pool* tmp;
68    gp.get_extension(tmp);
69    return tmp;
70  }
71
72  //Die gleiche Funktion wie oben, aber mit Hilfe der Erweiterungs-API
73  test_extension_pool* get_api(tlm::tlm_generic_payload& gp)
74  {
75    return catlm::extension_api::get<test_extension_pool>(gp);
76  }
77
78  //Teste die Erweiterung auf Gueltigkeit
79  bool test_pooled(tlm::tlm_generic_payload& gp)
80  {
81    test_extension_pool* tmp;
82    gp.get_extension(tmp);
83    if (tmp)
84      return tmp->is_valid;
85    else
86      return false;
87  }
88
89  //Die gleiche Funktion wie oben, aber mit Hilfe der Erweiterungs-API
90  bool test_api(tlm::tlm_generic_payload& gp)
91  {
92    return catlm::extension_api::is_valid<test_extension_pool>(gp);
93  }
94
95  //Ueberpruefe auf Gueltigkeit und rufe dabei auch die Erweiterung ab
96  bool test_n_get_pooled(tlm::tlm_generic_payload& gp, test_extension_pool*& ext)
97  {
98    gp.get_extension(ext);
99    if (ext) return ext->is_valid;
100   else return false;
101 }
102
103 //Die gleiche Funktion wie oben, aber mit Hilfe der Erweiterungs-API
104 bool test_n_get_api(tlm::tlm_generic_payload& gp, test_extension_pool*& ext)
105 {
106   return catlm::extension_api::test_and_get<test_extension_pool>(gp,ext);
107 }
```

Listing G.1: Zugriffsfunktionen für eine Pool-basierte Auto-Erweiterung

Es wurden fünf Testläufe pro Erweiterungskonzept (einzelne Auto-Erweiterung oder das in Abschnitt 4.8 vorgeschlagene Konzept) durchgeführt:

Speicherverwaltung : Bei diesem Test wird ein GP aus einem Pool entnommen und der Referenzzähler auf Eins gesetzt. Mittels `val_n_get_[pooled|api]` (siehe Listing G.1) wird eine Erweiterung hinzugefügt und als gültig markiert. Danach wird der Referenzzähler dekrementiert, sodass das GP in den Pool zurückkehrt, wobei alle Auto-Erweiterungen entfernt werden. Dieser Ablauf wird 10 Millionen Mal wiederholt[1]. Bei diesem Test können die Kosten des Hinzufügens einer Erweiterung, sowie die Kosten beim Wiedereinfügen eines GPs in den Pool untersucht werden.

Gültigkeitsänderung : Bei diesem Test wird ein GP aus einem Pool entnommen und der Referenzzähler auf Eins gesetzt. Mittels `val_n_get_[pooled|api]` wird eine Erweiterung hinzugefügt und als gültig markiert. Danach soll 10 Millionen Mal die Gültigkeit geändert werden. Da aber `validate` aus Abbildung 4.38 und `revalidate_pool` unabhängig davon sind, ob die Erweiterung bereits gültig war oder nicht, können die 10 Millionen Änderungen, durch das wiederholte Markieren der Gültigkeit emuliert werden. Abschliessend wird der Referenzzähler des GP dekrementiert und es kehrt in den Pool zurück. Bei diesem Test wird der Performanceunterschied zwischen dem Ändern eines `bool` Attributs einer Erweiterung und dem Setzen eines globalen, statischen Zeigers in das Erweiterungsfeld des GPs vermessen.

Erweiterungsabruf : Bei diesem Test wird ein GP aus einem Pool entnommen und der Referenzzähler auf Eins gesetzt. Mittels `val_n_get_[pooled|api]` wird eine Erweiterung hinzugefügt und als gültig markiert. Danach wird 10 Millionen Mal die Erweiterung mittels `get_[pooled|api]` aus dem GP abgerufen. Abschliessend wird der Referenzzähler des GP dekrementiert und es kehrt in den Pool zurück. Bei diesem Test wird vermessen, welche Kosten das Abfragen und ggf. Hinzufügen der Sticky-Erweiterung verursachen.

Gültigkeitsüberprüfung : Bei diesem Test wird ein GP aus einem Pool entnommen und der Referenzzähler auf Eins gesetzt. Mittels `val_n_get_[pooled|api]` wird eine Erweiterung hinzugefügt und als gültig markiert. Danach wird 10 Millionen Mal die Erweiterung mittels `test_[pooled|api]` auf Gültigkeit überprüft. Abschliessend wird der Referenzzähler des GP dekrementiert und es kehrt in den Pool zurück. Bei diesem Test wird der Performanceunterschied zwischen dem kombinierten Existenz- und `bool`-Attribut-Test (bei der Auto-Erweiterung) und dem Vergleich mit einem globalen, statischen Zeiger (beim vorgeschlagenen Konzept) vermessen.

Kombination Gültigkeitsüberprüfung und Erweiterungsabruf : Bei diesem Test wird ein GP aus einem Pool entnommen und der Referenzzähler auf Eins gesetzt. Mittels `val_n_get_[pooled|api]` wird eine Erweiterung hinzugefügt und als gültig markiert. Danach wird 10 Millionen Mal die Erweiterung mittels `test_n_get_[pooled|api]` auf Gültigkeit überprüft und abgerufen. Abschliessend wird der Referenzzähler des GP dekrementiert

[1] Diese hohe Zahl begründet sich damit, dass so die Funktion im Viewer schneller auffindbar ist.

und es kehrt in den Pool zurück. Bei diesem Test wird der Performanceunterschied bezüglich des Zugriffs auf nur eine Erweiterung (Auto-Erweiterung) im Vergleich zu zwei Erweiterungen (vorgeschlagenes Konzept) vermessen.

Neben diesen Tests bezüglich der Erweiterungsperformance wurde ein weiterer Test durchgeführt. Dabei wurde die Kosten zur Berechnung der nächsten Adresse eines OCP-Wrap-Bursts vermessen. Dazu wurde der im kommerziellen OCP-Channel-Kit [OCP-08]$_i$ verfügbare Code benutzt. Dieser Code bietet eine Klasse `BurstSequence` an, mit deren Funktion `next()` die nächste Addresse eines Bursts berechnet werden kann. Die Klasse muss nur zu Beginn eines Bursts initialisiert werden.

G.3. Ergebnisse

Die Testläufe wurden wie oben beschrieben durchgeführt. Dabei wurden die Kosten der Zugriffe in Form von ausgeführten Prozessorinstruktionen mit Hilfe von `callgrind` [Weid06]$_i$ vermessen. Der Testrechner entspricht dem in Abschnitt 5.2. Die Ergebnisse sind in Tabelle G.1 aufgelistet.

G.4. Auswertung

Wie erwartet ist die Speicherverwaltung bei Verwendung der eigentlichen Erweiterung als Sticky-Erweiterung und zusätzlicher Schalterweiterung effizienter als bei einer einzelnen Auto-Erweiterung. Grund dafür sind die eingesparten Zugriffe auf den Pool. Gültigkeitsänderungen und Gültigkeitstests sind bei beiden Ansätzen in etwa gleich schnell. Beim Abruf bzw. der Kombination aus Abruf und Gültigkeitsüberprüfung ist die Performance der einzelne Auto-Erweiterung mit 8 Instruktionen doppelt so schnell wie das vorgeschlagene Vorgehen (16 Instruktionen). Im Verlauf einer Transaktion wird die Speicherverwaltung nur einmal aktiv. Geht man davon aus, dass eine Erweiterung öfter abgerufen als gesetzt wird, dominiert also die schlechtere Performance des Abrufs bei Verwendung des vorgeschlagenen Vorgehens sehr schnell die Gesamtperformance der Erweiterungszugriffe. Konstruiert man den (unrealistischen) Fall von Millionen von Erweiterungsabrufen während einer Transaktion und enthält die Simulation kein anderes Verhalten als die Abrufe allein, so wird die Simulationszeit mit dem vorgeschlagenen Vorgehen in etwa doppelt so lange sein, als bei Verwendung der einzelnen Auto-Erweiterung, da das Zwei-zu-Eins-Verhältnis beim Erweiterungsabruf dann überwiegt.

Testlauf	Vermessene Funktion	Gesamtkosten	Einzelkosten	Kostendifferenz: vorgeschlagenes Konzept zur einzelnen Auto-Erweiterung
Speicherverwaltung	`val_n_get_pooled`	440.002.377	≈44	-4
	`val_n_get_api`	400.000.284	≈40	
	`test_extension_pool::free()`	180.000.000	18	-14
	`guard_of<test_extension_pool>::free()`	40.000.000	4	
Gültigkeitsänderung	`revalidate_pooled`	200.000.000	20	-1
	`revalidate_api`	190.000.000	19	
Erweiterungsabruf	`get_pooled`	80.000.000	8	+8
	`get_api`	160.000.000	16	
Gültigkeitsüberprüfung	`test_pooled`	120.000.000	12	-1
	`test_api`	110.000.000	11	
Kombination	`test_n_get_pooled`	140.000.000	14	+8
	`test_n_get_api`	220.000.000	22	
Addressberechnung	`BurstSequence<sc_dt::uint64>::next()`	2.006.000.000	200	nicht anwendbar

Tabelle G.1.: Messergebnisse zur Erweiterungsperformance

Ist also das vorgeschlagene Vorgehen zu langsam? Dazu müssen die Zahlen in realen Kontext gesetzt werden. Ein Modul greift nicht nur auf eine Erweiterung zu. Abhängig vom Ergebnis des Zugriffs wird es verschiedene Aktionen ausführen. Dazu kommen noch Aktionen, die unabhängig von der Erweiterung bei Empfang einer Transaktion durchzuführen sind. Als repräsentatives Beispiel für eine solche Aktion soll hier die Berechnung der Adressfolge eines OCP-Wrap-Bursts gelten. Diese Aktion muss bei jeder neuen Phase vom Initiator und Target ausgeführt werden. Interconnects werden dies in der Regel nicht tun, müssen dafür aber Einträge in Routing-Tabellen abfragen, Arbitrierungsentscheidungen treffen, Time-Outs kontrollieren usw. Der Aufwand dafür kann mit dem der Berechnung der Folgeadresse verglichen werden. Tabelle G.1 zeigt, dass für diese Berechnung 200 Instruktionen notwendig sind. Nimmt man an, dass nur eine Erweiterung parallel dazu abzufragen ist, ergeben sich dann 208 Instruktionen bei einer einzelnen Auto-Erweiterung und 216 Instruktionen beim vorgeschlagenen Vorgehen. Der Unterschied liegt dann nur noch bei 3,8%.

In Modellen real-existierender IP wird bei einer Kommunikation meist noch deutlich mehr berechnet als Folgeadressen. Die Anzahl der Instruktionen dafür erreich den vierstelligen Bereich, sodass der Overhead durch das vorgeschlagende Vorgehen dann selbst bei Verwendung vieler Erweiterungen unterhalb von einem Prozent liegt. So benötigen zum Beispiel die Prozesse `enqueue_req` und `ereq_handler` aus Abbildung 6.6 auf Seite 165 jeweils circa 900 Instruktionen, der Porzess `arbitrate` circa 600 und der Prozess `req_deliverer` sogar 1240 Instruktionen[2].

Darüber hinaus, wird auch deutlich, dass die effiziente Speicherverwaltung beim vorgeschlagenen Konzept die Mehrkosten der ersten zwei Erweiterungsabfragen ausgleicht. Nur zwei Abfragen einer Erweiterung pro Transaktion treten z.B. bei einer Erweiterung mit x2x-Änderungsintervall auf einem Transaktionspfad vom Initiator über ein Interconnect zum Target auf. Dort werden nur Interconnect und Target die Erweiterung je einmal abfragen. Dies ist ein recht häufiger Fall. Im PLB-Interface aus Abschnitt 6.2 zum Beispiel sind zehn von 17 Erweiterung mit einem x2x-Ändderungsintervall ausgestattet.

In Anbetracht der am Ende von Abschnitt 4.8.4 aufgelisteten Vorteile der vorgeschlagenen Vorgehens, ist diese geringe Performanceeinbuße vertretbar.

[2]Dies wurde im Rahmen der Entwicklung des PLB-Modells aus Abschnitt 6.3.1 vermessen.

H. Interfaces zwischen Takt und Taktsynchronisierer

Inhalt

H.1. Einleitung

In Abschnitt 4.10.3 wurden unter anderem Interfaces zwischen Quell-Takten und Taktsynchronisierern und abgeleiteten Takten und Taktsynchronisierern. eingeführt. Dort wurden ihre Funktionen nur sehr knapp beschrieben. Dieser Anhang listet die einzelnen Funktionen nochmals auf und erklärt sie im Detail.

H.2. `tlm_clock_sync_src_if`

`void src_stop(const sc_time& last_edge, const sc_time& per_at_stop)` :

Wann immer ein Takt stoppt, ruft er diese Funktion (ggf. auf dem Default-Synchronisierer) auf. Dabei übergibt er als erstes Argument, die Zeitmarke zu der die letzte sichtbare Taktflanke des Taktes auftrat. Diese Zeitmarke darf bis zu einer Taktperiode in der simulierten Vergangenheit liegen. Das zweite Argument bestimmt die Taktperiode, die für den Taktzyklus gilt, der mit der letzten sichtbaren Flanke startete. Der Default-Synchronisierer muss daraufhin nichts tun.

`void src_restart()` :

Wann immer ein Takt wieder startet, ruft er diese Funktion (ggf. auf dem Default-Synchronisierer) auf. Der Aufruf muss zu der Simulationszeit erfolgen, bei der die erste (neue) Flanke des Taktes auftritt. Der Default-Synchronisierer muss daraufhin nichts tun.

`void src_period_change()` :

Bevor ein Takt seine Frequenz ändern will, ruft er diese Funktion (ggf. auf dem Default-Synchronisierer) auf. Der Aufruf erfolgt zum gleichen Simulationszeitpunkt wie eine Taktflanke, aber bevor die Taktflanke ausgelöst wird. Wichtig dabei ist, dass die Zugriffe auf die Periode des Quelltaktes (siehe unten) die Werte liefern, als hätte die Änderung noch nicht stattgefunden, damit der Synchronisierer noch Zugriff auf die alten Werte hat. Das bedeutet, der Aufruf markiert lediglich, dass eine Periodenänderung unmittelbar bevor steht. Der Default-Synchronisierer muss daraufhin nichts tun.

`void src_set_clock(tlm_clock_sync_access_src_if*)` :

Diese Funktion wird aufgerufen, um das Klassenattribut `m_src` auf den übergebenen Wert zu setzen, dem Synchronisierer also den Takt zu übergeben, der der Quelltakt ist.

H.3. `tlm_clock_sync_dst_if`

`void dst_stop(const sc_time&)` :

Wann immer ein Takt zwischen einem Aufruf von `stop()` und einem Aufruf von `start()` stoppt, ruft er diese Funktion (ggf. auf dem Default-Synchronisierer) auf[1]. Der Default-Synchronisierer muss daraufhin nichts tun. Wird auf einem abgeleiteten Takt `stop()` aufgerufen, während sein Quelltakt gestoppt ist, so darf `dst_stop()` nur aufgerufen werden, wenn der Quelltakt wieder läuft und der abgeleitete Takt nach wie vor gestoppt ist.

`void dst_restart()` :

Wann immer ein Takt wieder startet, ruft er diese Funktion (ggf. auf dem Default-Synchronisierer) auf. Der Aufruf muss zu der Simulationszeit erfolgen, bei der die erste (neue) Flanke des Taktes auftritt. Der Default-Synchronisierer muss daraufhin nichts tun.

`void dst_period_change(const sc_time& new_period)` :

Bevor ein Takt seine Frequenz ändert, ruft er diese Funktion auf und übergibt als Argument die Periode, die er als neue Periode zu verwenden beabsichtigt. So kann der Synchronisierer diese ggf. korrigieren. Der Default-Synchronisierer braucht bei diesem Aufruf nichts zu tun.

`sc_event* dst_request_start_event()` :

Bevor ein Takt startet, fordert er vom Synchronisierer ein Event an, auf das er wartet, bevor er wirklich startet. Liefert der Aufruf einen `NULL`-Zeiger zurück, so wartet der Takt nicht. Der Default-Synchronisierer muss auf diesen Aufruf hin `NULL` zurückliefern.

[1] Zur Verdeutlichung: Ein Quelltakt teilt jeden Stopp unabhängig von der Stopp-Ursache mittels `src_stop()` mit, während ein abgeleiteter Takt nur die Stopps mitteilt, die durch Starts und Stopps mit Hilfe des `tlm_clock_if` entstehen.

`sc_time dst_get_period()` :

Wird ein Takt mittels `period()` aus dem `tlm_clock_if` nach seiner Periode gefragt, so ruft er stets diese Funktion (ggf. auf dem Default-Synchronisierer) auf. Liefert diese Funktion `SC_ZERO_TIME` zurück, so ist der Takt ein Quelltakt und muss den Rückgabewert für `period()` selbst bestimmten. Anderenfalls handelt es sich um einen abgeleiteten Takt und der Rückgabewert von `dst_get_period()` ist auch der Rückgabewert für `period()`. Der Default-Synchronisierer muss dementsprechend stets `SC_ZERO_TIME` zurückliefern.

`void dst_set_clock(tlm_clock_sync_access_src_if*)` :

Diese Funktion wird aufgerufen, um das Klassenattribut `m_dst` auf den übergebenen Wert zu setzen, dem Synchronisierer also den Takt zu übergeben, der der abgeleitete Takt ist.

H.4. tlm_clock_sync_access_src_if

`sc_time& get_local_time()` :

Dieser Aufruf liefert die Zeitmarke zurück, bis zu der der Takt alle benötigten Flanken bereits erzeugt hat. Diese Zeitmarke darf bis zu einer Taktperiode von der aktuellen Simulationszeit in der Zukunft liegen.

`const sc_time& period() const` :

Dieser Aufruf liefert die aktuelle (also zuletzt zugewiesene) Periode des Taktes zurück. Dies ist nicht notwendigerweise die Periode die zurzeit angewendet wird[2].

`const sc_time& effective_period() const` :

Dieser Aufruf liefert die zurzeit Anwendung findende Periode zurück.

`sc_event& edge_in_cycle(unsigned latency)` :

Dieser Aufruf entspricht dem aus dem `tlm_clock_interface`.

`sc_time start_time() const` :

Dieser Aufruf entspricht dem aus dem `tlm_clock_interface`.

H.5. tlm_clock_sync_access_dst_if

`void request_restart()` :

Dieser Aufruf fordert den Takt auf zu starten, wenn keine anderen Bedingungen den Takt davon abhalten (wie z.B. das direkte Stoppen des Taktes über das `tlm_clock_if`).

[2]Nach der Zuweisung einer Periode kann ein Takt z.B. diese erst beim Auftreten der nächsten Flanke, welche noch nach Ablauf der alten Periode auftrat, übernehmen.

`void request_stop()` :

Dieser Aufruf weist den Takt an zum nächstmöglichen Zeitpunkt anzuhalten. Welcher genau das ist, legt die Implementierung des Taktes fest.

`bool request_period_invalidate()` :

Dieser Aufruf informiert den Takt, dass die Quelltaktperiode gewechselt hat und somit die Periode des abgeleiteten Taktes geändert werden muss. Kann der abgeleitete Takt direkt zum Aufrufzeitpunkt seine Periode ändern, so kann er mittels `dst_get_peroid` die neue Periode abfragen und setzen. In diesem Fall muss `true` zurückgeliefert werden. Kann die Periode nicht gewechselt werden, weil z.B. die Änderung mitten in einem Takt erfolgt, so ist `false` zruückzuliefern. In diesem Fall muss der Synchronisierer den abgeleiteten Takt stoppen und zu einem geeigneten Zeitpunkt neu starten.

`sc_time& get_local_time()` :

Dieser Aufruf entspricht dem aus dem `tlm_clock_sync_access_src_if`.

`sc_event& posedge_in_cycle(unsigned latency)` :

Dieser Aufruf entspricht dem aus dem `tlm_clock_interface`.

I. AMBA AHB: Formale Erfassung der Busphasen

Inhalt

I.1. Einleitung

Im Gegensatz zu AXI und APB unterscheiden sich die Interfaces von Master und Slave im AHB. Abbildung I.1, Seite 239, zeigt beide Interfaces[1]. Die Interfaces unterscheiden sich zuerst im Arbitrierungsbus: Der Master kann dort den Bus anfordern (HBUSREQ), ihn zugeteilt bekommen (HGRANT) und beim Anfordern auch verriegeln (HLOCK). Der Slave dagegen erhält über den Arbitrierungsbus die Information welcher Master zurzeit den Addressbus besitzt (HMASTER) und ob die Arbitrierung aktuell verriegelt ist (HMASTLOCK). Darüber hinaus kann der Slave den Bus auffordern, einen mittels einer sogenannten

[1] Hinweis: Die Gruppierung der Signale in Arbitrierungs-, Address- und Datenbus ist von mir gewählt und nicht offizieller Teil der AHB-Spezifikation

Split-Response von der Arbitrierung ausgeschlossenen Master wieder in die Arbitrierung aufzunehmen (HSPLIT).

Die Addressbusse von Master und Slave sind nahezu identisch; einziger Unterschied ist die Information für den Slave, ob er aktuell angesprochen wird (HSEL). Die weiteren Signale des Addressbusses übermitteln, ob ein Transfer läuft (HTRANS), ob es ein Lese- oder Schreibtransfer ist (HWRITE), zu welcher Addresse der Transfer geht (HADDR) und noch weitere, für die folgenden Betrachtungen nicht relevante, zusätzliche Transferqualifizierer (HSIZE, HBURST, HPROT).

Auch die Datenbusse sind nahezu identisch. Hier besitzt der Salve zusätzlich zum globalen Ready-Signal (HREADY beim Master und HREADYIN beim Slave) noch einen eigenen Ready-Ausgang (HREADYOUT). Die gleichen Signale sind die Schreibdaten (HWDATA), die Lesedaten (HRDATA), die Transferantwort (HRESP) und das bereit erwähnte globale Ready-Signal.

Der in Abbildung I.1 gezeigte Ablauf präsentiert ein 4-Wort Schreib-Burst von Master 1 an den Slave, der Adressen A0 bis (mindestens) A3 bedient, dabei sind die dargestellten Master- und Slave-Interfaces die Interfaces dieser beiden Module. Dazwischen befindet sich dann ein AHB-Bus. Zum grundlegenden Verständnis der Festlegung der Busphasen ist eine knappe Erläuterung der AHB-Funktion notwendig. Der AHB kennt vier Transfermodi: IDLE, BUSY, NONSEQ, und SEQ, welche über HTRANS bekannt gegeben werden. IDLE bedeutet, dass der Master jetzt keinen Transfer durchführen will und ggf. ein bis jetzt noch nicht beendeter Burst nun abgebrochen wird. BUSY bedeutet, dass der Master jetzt keinen Transfer durchführen will aber ein bis jetzt noch nicht beendeter Burst demnächst fortgesetzt wird. NONSEQ bedeutet, dass der Master jetzt einen Transfer durchführen will und dass dies der erste Transfer eines neuen Bursts oder ein neuer Einzelworttransfer ist; ein bis jetzt noch nicht beendeter Burst wird dadurch abgebrochen. SEQ bedeutet, dass der Master einen Transfer durchführen will und das es sich dabei um den nächsten Transfer innerhalb eines Bursts handelt.

Ein Transfer beim AHB besteht immer aus einem Adresszyklus und einem Datenzyklus. Im Addresszyklus wird der Slave identifiziert, der den Datenzyklus bearbeiten soll. Ein Adresszyklus startet in einem Takt, wenn im vorangegangenen Takt das globale HREADY-Signal auf Eins war und er endet wenn es wieder auf Eins ist. Dabei können Start und Ende auf den gleichen Takt fallen. Der zu einem Addresszyklus gehörende Datenzyklus startet immer, wenn im vorangegangenen Takt der Adresszyklus endete und endet, wenn das globale HREADY auf Eins ist. In Abbildung I.1 z.B. startet ein SEQ-Addresszyklus (Adresse A1) in Takt 5 und endet in Takt 6 und der zugehörige Datenzyklus (Daten D1) startet in Takt 7 und endet in Takt 8.

Die Flußkontrolle für beide Zyklen ist also durch das globale HREADY gegeben, welches aber stets dem HREADY des aktuell im Datenzyklus befindlichen Slave entspricht. Das bedeutet, dass die Dauer eines Addresszyklus eines NONSEQ-Transfers potentiell durch einen anderen Slave (nämlich den, der im vorherigen Addresszyklus aktiviert wurde und nun

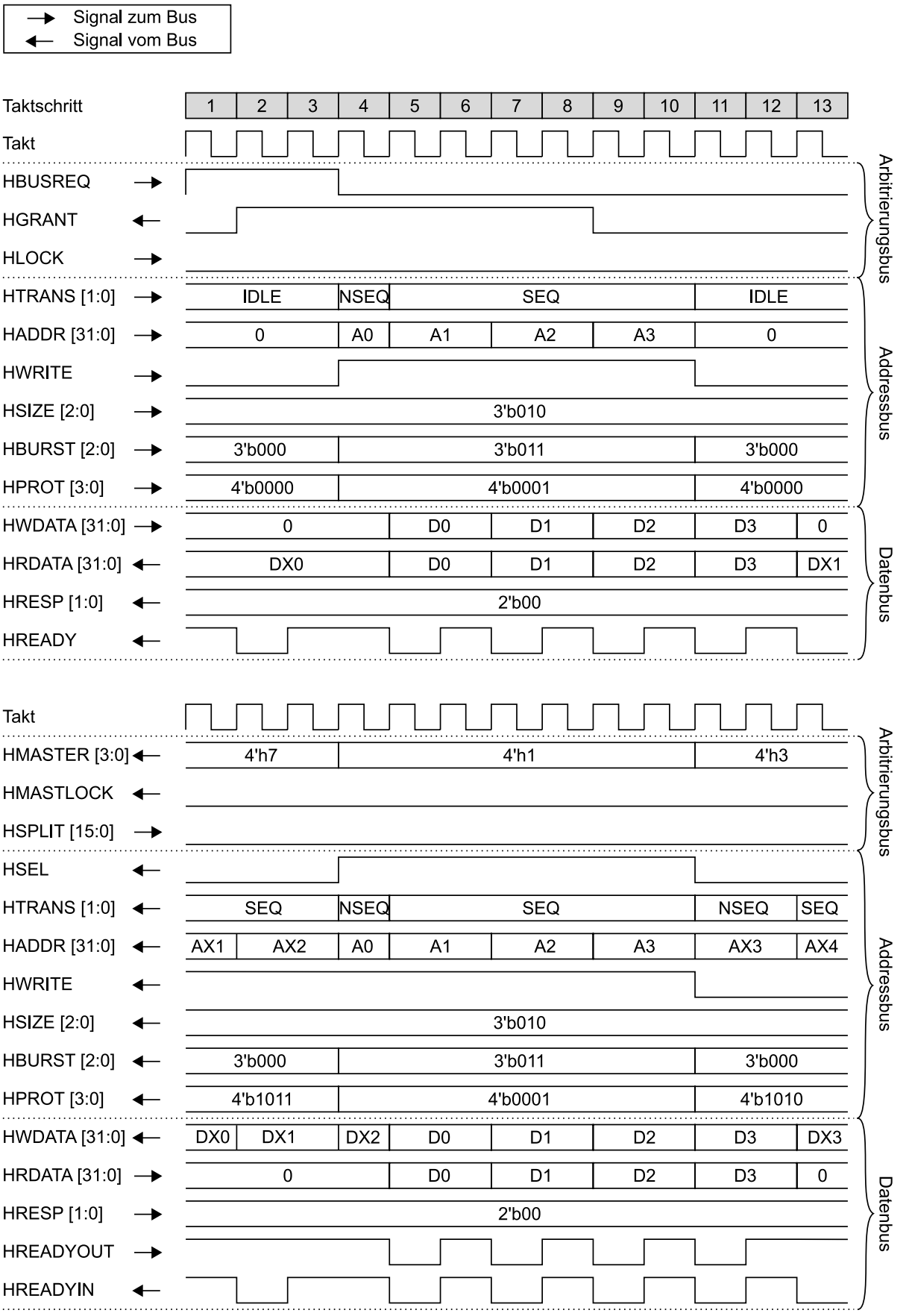


Abbildung I.1.: Beispielabläufe an AHB-Master- und -Slave-Interfaces

im Datenzyklus ist) als den aktuell adressierten Slave gesteuert wird. Der aktuell adressierte Slave hat also bei NONSEQ-Transfers grundsätzlich keine Flußkontrolle. Dementsprechend gilt laut AHB-Spezifikation, dass ein Slave nur wenn das globale HREADY auf Eins ist, die Signale des Adressbus auswerten darf, aber eben auch auswerten muss.

Für jeden AHB-Slave muss laut AHB-Spezifikation die maximale Anzahl von Takten, in denen der Slave HREADY auf 0 setzt, also das Ende eines Datenzyklus verzögert, angegeben werden. Die größte solche Verzögerung im zu modellierenden System nenne ich im Folgenden max. Die Existenz dieser maximalen Verzögerung wird es erlauben, effiziente Phasendefintionen zu finden.

I.2. Busphasen im Master-Interface

Die Interfaces aller Master in einem AHB-System sind identisch. Somit genügt es einen Satz Busphasen für einen Master zu definieren. Dieser Satz kann dann unter Umbenennung der masterspezifischen Signale für jeden Master angenommen werden. Die Phasen werden nun gemäß Definition 3.7 festgelegt. Die Portzuordnung $pz(m)$ für den exemplarischen Master m ergibt sich in trivialer Weise aus den in Abbildung I.1 gezeigten Signalen des Master-Interfaces, da Portname, Bitbreite und Richtung dargestellt sind. Ein Port aus dieser Portzuordnung ist per Definition 3.1 z.B. $("HBUSREQ", 1, e)$. Zur Verkürzung der Folgenden Notationen, sei für ein solches Tupel stellvertretend nur der Portname (im Beispiel also nur HBUSREQ) genannt, da über diesen eine eindeutige Zuordnung zum eigentlichen Tupel möglich ist.

Port	HBUSREQ	HGRANT	HLOCK	HTRANS	HADDR	HWRITE	HSIZE
Default	0	0	0	IDLE	0	0	0

Fortsetzung						
Port	HBURST	HPROT	HWDATA	HRDATA	HRESP	HREADY
Default	0	0	0	0	0	1

Tabelle I.2.: Defaultzustände für die Ports im AHB-Master-Interface

Tabelle I.2 definiert die Defaultzustände der Ports des Master-Interfaces. Sind alle Signale im Defaultzustand so finden im Master-Interface ununterbrochen IDLE-Transfers ab, es wird also nicht kommuniziert.

Tabelle I.3 zeigt die Phasenports ($PP_\Phi = PP_{\Phi S} \cup PP_{\Phi E}$), Observierungsports ($PP_{\Phi O}$) und Beobachtungsdauern (tb_Φ) der Busphasen im Master-Interface und benennt ihre Typen gemäß Anhang B. Die Bedeutungen der Phasen werden im Folgenden erläutert und danach dann die Start-, Ende und Abbruchkriterien bestimmt und gegebenenfalls werden Besonderheiten hervorgehoben. Zur Straffung der Notation wird ein Hilfskonstrukt benötigt:

$$LR = \{ \quad ((w_1, ..., w_{max+1}), n) \in (SI_{HREADY})^{max+1} \times \mathbb{N} \mid$$
$$n \leq max + 1 \wedge w_n = 1 \wedge (\forall i \in \mathbb{N}, n < i \leq max + 1 : w_i = 0)\}$$

Ein Paar (w, n), bestehend aus einem $max + 1$ Takte langen Signalablauf von HREADY w und einem Index n dieses Teilabschnittes, ist genau dann in LR (LR steht für „letztes Ready“), wenn in Takt n HREADY das letzte Mal auf Eins und danach nur noch auf Null war. In jedem AHB-spezifikationskonformen System kann zu jedem beliebigen $max+1$ Takte lange Teilabschnitt w des gesamten Signalablaufs von HREADY genau ein n gefunden werden, sodass $(w, n) \in LR$ gilt, da wie oben beschrieben HREADY maximal max Takte lang Null sein darf.

	PP_Φ				
Phasenname	$PP_{\Phi S}$	$PP_{\Phi E}$	$PP_{\Phi O}$	tb_Φ	Typ
BUSREQ	{HBUSREQ}	$\emptyset$	$\emptyset$	2	Multitakt
GRANT	{HGRANT}	$\emptyset$	{HREADY}	$max + 2$	Multitakt
LOCK	{HLOCK}	$\emptyset$	$\emptyset$	2	Multitakt
REQ	{HTRANS, HADDR, HWRITE, HSIZE, HBURST, HPROT, HREADY}	{HREADY}	$\emptyset$	2	Multitakt
BURSTREQ	{HTRANS, HADDR, HWRITE, HSIZE, HBURST, HPROT}	$\emptyset$	{HREADY}	2	Multitakt
DATA	{HWDATA, HREADY}	{HREADY}	{HTRANS, HWRITE}	$max + 2$	Multitakt
RESP	{HRDATA, HREADY}	$\emptyset$	{HTRANS, HWRITE}	$max + 2$	Einzeltakt
NOTOK	{HRESP, HREADY}	$\emptyset$	{HTRANS}	$max + 2$	Multitakt

Tabelle I.3.: Phasenstartports, Phasenendports, Beobachtungsdauern und Phasentypen im AHB-Master-Interface

BUSREQ und LOCK Der Kommunikationsautomat muss zu jedem Takt über den Zustand der Busanfragen aller Master in Kenntnis sein, um korrekt arbitrieren zu können. Dementsprechend werden die BUSREQ- bzw. LOCK-Phase immer starten, sobald HBUSREQ bzw. HLOCK gesetzt wird und abbrechen wenn HBUSREQ bzw. HLOCK wieder zurückgesetzt

wird. Da keine Eingangssignale des Masters an den Phasen beteiligt sind, gibt es auch kein reguläres Ende der Phasen. Somit gilt

$S_{BUSREQ} = \{(w_1, w_2) \in (SI_{HBUSREQ})^2 | w_1 = 0 \wedge w_2 = 1\}$

$E_{BUSREQ} = \emptyset,$

$A_{BUSREQ} = \{(w_1, w_2) \in (SI_{HBUSREQ})^2 | w_1 = 1 \wedge w_2 = 0\}$

$S_{LOCK} = \{(w_1, w_2) \in (SI_{HLOCK})^2 | w_1 = 0 \wedge w_2 = 1\}$

$E_{LOCK} = \emptyset$

$A_{LOCK} = \{(w_1, w_2) \in (SI_{HLOCK})^2 | w_1 = 1 \wedge w_2 = 0\}$

GRANT Ein Master darf auf den AHB zugreifen, wenn im vorangegangenen Takt HGRANT und HREADY auf Eins waren. Er verliert das Zugriffsrecht, wenn HGRANT auf Null und HREADY auf Eins ist. Änderungen von HGRANT während HREADY auf NULL ist, können ignoriert werden. Es bietet sich an, eine Multitaktphase zu definieren, die startet, wenn HREADY und HGRANT beide auf Eins sind und die endet wenn HGRANT Null und HREADY Eins ist. Dies bedeutet, dass aus Sicht dieser Phase zwischen Start und Ende Änderungen der Signale unterdrückt sind. Betrachtet man Abbildung I.1 würde die Phase in Takt 3 starten und in Takt 10 enden und dabei die Änderungen in HGRANT in Takt 2 bzw. 9 unterdrücken (bzw. verschieben). Das ist akzeptabel, da es an der Arbitrierungsentscheidung nichts ändert. Jedoch ändert sich auch HREADY zwischen Takt 3 und 10 häufig. Ein Unterdrücken dieser Änderungen ist nicht akzeptabel, da HREADY die Address- und Datenzyklen steuert. Aus diesem Grund wird HREADY zu den Phasenobservierungsports und nicht zu den Phasenstartports gezählt. Dies erlaubt, HREADY zur Bestimmung von Phasenstart und Abbruch heranzuziehen, ohne den Wert aus Sicht der GRANT-Phase jemals festzulegen.

$$S_{GRANT} = \{ \quad (w_1, ..., w_{max+2}) \in \left(\bigtimes_{p \in PP_{GRANT} \cup PP_{GRANTO}} SI_p \right)^{max+2} \quad | \\ proj_{HREADY}(w_{max+2}) = 1 \wedge proj_{HGRANT}(w_{max+2}) = 1 \wedge \\ [\exists n \in \mathbb{N}, 1 \leq n \leq max + 1 : \\ (proj_{HGRANT}(w_n) = 0 \wedge ((proj_{HREADY}(w_\nu))_{\nu=1...max+1}, n) \in LR)]\}$$

Das Startkriterium kann gelesen werden als: Eine Phase startet, wenn HREADY und HGRANT auf Eins sind und wenn beim letzten HREADY davor HGRANT auf Null war, es also eine sichtbare (also bei HREADY=1 geschehene) Änderung gab. Da es mindestens einen Takt mit HREADY=1 in den $max + 1$ Takten vor dem aktuellen Takt gegeben haben muss, impliziert eine Auswertung des in eckigen Klammern stehenden Terms zu „nicht wahr“, dass beim letzen HREADY auch HGRANT auf Eins war und daher die Phase nicht starten muss (sie ist schon gestartet). Das Endkriterium ist leer und das Abbruchkriterium ergibt sich analog zum Startkriterium.

$E_{GRANT} = \emptyset,$

$$A_{GRANT} = \{ \quad (w_1, ..., w_{max+2}) \in \left(\bigtimes_{p \in PP_{GRANT} \cup PP_{GRANTO}} SI_p \right)^{max+2} \mid$$
$$proj_{HREADY}(w_{max+2}) = 1 \wedge proj_{HGRANT}(w_{max+2}) = 0 \wedge$$
$$[\exists n \in \mathbb{N}, 1 \leq n \leq max + 1 :$$
$$(proj_{HGRANT}(w_n) = 1 \wedge ((proj_{HREADY}(w_\nu))_{\nu=1...max+1}, n) \in LR)]\}$$

REQ und BURSTREQ Die Phase REQ wird schlicht einem NONSEQ-Addresszyklus entsprechen, während die BURSTREQ-Phase die SEQ- und BUSY-Addresszyklen abdeckt. Prinzipell könnten beide Phasen auch durch eine einzige Phase abgedeckt werden, jedoch ist diese Aufteilung im Slave-Interface unvermeidbar (sie wird dort begründet) und wird aus Symetriegründen auch im Master-Interface beibehalten.

$$S_{REQ} = \{ \quad (w_1, w_2) \in \left(\bigtimes_{p \in PP_{REQ}} SI_p \right)^2 \mid$$
$$proj_{HTRANS}(w_2) = NONSEQ \wedge$$
$$[proj_{HTRANS}(w_1) \neq NONSEQ \vee$$
$$(proj_{HTRANS}(w_1) = NONSEQ \wedge proj_{HREADY}(w_1) = 1)]\}$$

$$E_{REQ} = \{ \quad (w_1, w_2) \in \left(\bigtimes_{p \in PP_{REQ}} SI_p \right)^2 \mid$$
$$proj_{HTRANS}(w_2) = NONSEQ \wedge proj_{HREADY}(w_2) = 1\}$$

$$A_{REQ} = \{ \quad (w_1, w_2) \in \left(\bigtimes_{p \in PP_{REQ}} SI_p \right)^2 \mid$$
$$proj_{HTRANS}(w_1) = NONSEQ \wedge$$
$$proj_{HREADY}(w_1) = 0 \wedge$$
$$proj_{HTRANS}(w_2) = IDLE\}$$

$$S_{BURSTREQ} = \{ \quad (w_1, w_2) \in \left(\bigtimes_{p \in PP_{BURSTREQ}} SI_p \right)^2 \mid$$
$$[(proj_{HTRANS}(w_1) = NONSEQ \vee proj_{HTRANS}(w_1) = SEQ) \wedge$$
$$proj_{HTRANS}(w_2) = BUSY]$$
$$\vee$$
$$[(proj_{HREADY}(w_1) = 1 \vee proj_{HTRANS}(w_1) = BUSY) \wedge$$
$$proj_{HTRANS}(w_2) = SEQ]\}$$

$$E_{BURSTREQ} = \{ \quad (w_1, w_2) \in \left(\bigtimes_{p \in PP_{BURSTREQ}} SI_p \right)^2 \mid$$
$$proj_{HTRANS}(w_2) = SEQ \wedge proj_{HREADY}(w_2) = 1\}$$

$$A_{BURSTREQ} = \{ \quad (w_1, w_2) \in \left(\bigtimes_{p \in PP_{BURSTREQ}} SI_p \right)^2 \mid$$
$$[proj_{HTRANS}(w_2) = IDLE \vee proj_{HTRANS}(w_2) = NONSEQ] \wedge$$
$$[proj_{HTRANS}(w_1) = BUSY \vee$$
$$(proj_{HREADY}(w_1) = 0 \wedge proj_{HTRANS}(w_1) = SEQ)]\}$$

Man beachte, dass eine BURSTREQ-Phase nur im Falle eines SEQ-Addresszyklus endet. Eine BURSTREQ-Phase, die einen (oder mehr) BUSY-Addresszyklus modelliert, endet nie, sie kann lediglich abbrechen oder sie wird durch einen Neustart der BURSTREQ-Phase (dann als SEQ-Addresszyklus) ersetzt. Anderenfalls würde im Falle einer langen Verzögerung seitens des Masters (also viele BUSY-Addresszyklen) bei gleichzeitiger Empfangsbereitschaft des Slaves (also HREADY=1) die Phase in jedem Takt starten und enden, ohne dass dies wirklich Informationen trägt. Da sich aber während dieser Zeit HREADY ändern kann, darf HREADY nicht zu den Phasenports von BURSTREQ zählen, da ansonsten eine Änderung von HREADY während der BURSTREQ-Phase nicht zulässig wäre. Der Zustand von HREADY wird während der BURSTREQ-Phase von einer stets parallel laufenden DATA- oder RESP-Phase bestimmt.

DATA Die DATA-Phase entspricht einem NONSEQ- oder SEQ-Schreibdatenzyklus. Sie beginnt, sobald ein NONSEQ- oder SEQ-Schreib-Addresszyklus beendet ist (und somit die Schreibdaten gültig werden) und endet, wenn der Slave die Datenübernahme mit HREADY quittiert hat. Offenbar wird der Datenzyklus von Signalen, die nicht vom Datenzyklus steuerbar sind, kontrolliert. Dementsprechend werden diese Signale als Observierungsports verwendet.

$$S_{DATA} = \{ \quad (w_1, w_2) \in \left(\bigtimes_{p \in PP_{DATA} \cup PP_{DATAO}} SI_p \right)^2 \mid$$
$$(proj_{HTRANS}(w_1) = NONSEQ \vee proj_{HTRANS}(w_1) = SEQ) \wedge$$
$$proj_{HREADY}(w_1) = 1 \wedge proj_{HWRITE}(w_1) = 1\}$$

$$E_{DATA} = \{ \quad (w_1, ..., w_{max+2}) \in \left(\bigtimes_{p \in PP_{DATA} \cup PP_{DATAO}} SI_p \right)^{max+2} \mid$$
$$proj_{HREADY}(w_{max+2}) = 1 \wedge$$
$$[\exists n \in \mathbb{N}, 1 \leq n \leq max + 1 :$$
$$(proj_{HWRITE}(w_n) = 1 \wedge proj_{HTRANS}(w_n) \neq BUSY$$
$$\wedge proj_{HTRANS}(w_n) \neq IDLE \wedge$$
$$((proj_{HREADY}(w_\nu))_{\nu=1...max+1}, n) \in LR)]\}$$

$$A_{DATA} = \emptyset$$

Man sieht bei dieser Phase sehr gut, wie die Observierungssignale eigesetzt werden. Die Werte von HTRANS und HWRITE sind keinesfalls durch die Phase steuerbar, sie können jeden beliebigen Wert haben, aber der Wert in der Vergangenheit ist entscheidend (übermittel durch die REQ- bzw. BURSTREQ-Phase). Mit Hilfe der Observierungssignale wird

sicher gestellt, dass eine DATA-Phase nur in einem Takt nach einem akzeptierten NOSEQ- oder SEQ- Addresszyklus erfolgt und dass sie nur endet, wenn sie vorher auch gestartet ist.

RESP Diese Einzeltaktphase markiert den Takt, in dem ein NONSEQ- oder SEQ-Lese-Datenzyklus endet. Der Beginn eines solchen Datenzyklus muss nicht markiert werden, da dabei (im Gegensatz zum Schreib-Datenzyklus) keine Informationen übertragen werden.

$$
\begin{aligned}
S_{RESP} = \{ \quad & (w_1, ..., w_{max+2}) \in \left(\bigtimes_{p \in PP_{RESP} \cup PP_{RESPO}} SI_p \right)^{max+2} \quad | \\
& proj_{HREADY}(w_{max+2}) = 1 \wedge \\
& [\exists n \in \mathbb{N}, 1 \leq n \leq max + 1 : \\
& (proj_{HWRITE}(w_n) = 0 \wedge proj_{HTRANS}(w_n) \neq BUSY \wedge \\
& proj_{HTRANS}(w_n) \neq IDLE \wedge \\
& ((proj_{HREADY}(w_\nu))_{\nu=1...max+1}, n) \in LR)]\}
\end{aligned}
$$

$$E_{RESP} = S_{RESP}$$

$$A_{RESP} = \emptyset$$

NOTOK (lies „not Okay“) Per AHB-Protokoll haben Datenzyklen, die nicht mit HRESP=OK enden eine Zweitakt-Antwort[2]. Im ersten Takt der Antwort wechselt HRESP auf einen Wert ungleich OK, wobei HREADY Null sein muss und im zweiten Takt muss dann HREADY auf Eins gehen, um den Datenzyklus abzuschliessen. Dementsprechend startet die NOTOK-Phase im ersten Takt dieser Zweitakt-Antwort und endet im darauffolgenden Takt mit HREADY auf Eins.

$$
\begin{aligned}
S_{NOTOK} = \{ \quad & (w_1, ..., w_{max+2}) \in \left(\bigtimes_{p \in PP_{NOTOK} \cup PP_{NOTOKO}} SI_p \right)^{max+2} \quad | \\
& proj_{HRESP}(w_{max+2}) \neq OK \wedge \\
& proj_{HREADY}(w_{max+2}) = 0 \wedge [\exists n \in \mathbb{N}, 2 \leq n \leq max + 1 : \\
& (proj_{HTRANS}(w_n) \neq BUSY \wedge \\
& proj_{HTRANS}(w_n) \neq IDLE \wedge \\
& ((proj_{HREADY}(w_\nu))_{\nu=1...max+1}, n) \in LR)]\}
\end{aligned}
$$

Man beachte, dass n größer oder gleich Zwei sein muss. Der Grund dafür ist, dass der erste Takt der Zweitakt-Antwort HREADY=0 hat, also zu der maximal zulässigen Verzögerung hinzuzählt. Somit kann das letzte HREADY nur noch max, anstelle von $max + 1$ Takten in der Vergangenheit liegen.

$$E_{NOTOK} = \emptyset$$

[2]Bei HRESP=OK ist der Takt, in dem HREADY auf Eins ist und den Datenzyklus beendet, eine sogenannte Eintakt-Antwort.

$$A_{NOTOK} = \{ \quad (w_1, ..., w_{max+2}) \in \left(\bigtimes_{p \in PP_{NOTOK} \cup PP_{NOTOKO}} SI_p \right)^{max+2} \quad |$$
$$proj_{HRESP}(w_{max+2}) \neq OK \wedge proj_{HREADY}(w_{max+2}) = 1 \wedge$$
$$proj_{HREADY}(w_{max+1}) = 0 \wedge [\exists n \in \mathbb{N}, 1 \leq n \leq max + 1 :$$
$$(proj_{HTRANS}(w_n) \neq BUSY \wedge$$
$$proj_{HTRANS}(w_n) \neq IDLE \wedge$$
$$((proj_{HREADY}(w_\nu))_{\nu=1...max+1}, n) \in LR)]\}$$

Damit sind alle Phasen des Massters erfasst und es können die Phasen des Slaves festgelegt werden.

I.3. Busphasen im Slave-Interface

Die Interfaces von Slaves in einem AHB-System können sich unterscheiden. Es gibt sog. „Split-Capable“ Slaves und solche die es nicht sind. Erstere benutzen das Split-Transfer-Protokoll des AHB, letztere nicht. Das bedeutet, dass einige Slaves die einzig für das Split-Transfer-Protokoll vorgesehenen Ports HMASTLOCK, HSPLIT und HMASTER haben und andere nicht. Um das AHB-Protokoll also vollständig zu erfassen, muss die (ggf. virtuelle) Peripherie des AHB, die zur Festlegung der Busphasen verwendet wird, entweder beide Slave-Typen enthalten oder aber es müssen verschiedene AHBs, einer mit und einer ohne Split-Capable Slaves, mehrfach dem GP- und TLM-Phase-Mapping zugeführt werden.

Hier wird das mehrfache Mapping verwendet. Das bedeutet, zuerst wird ein AHB betrachtet, an dem ausschließlich Split-Capable Slaves angeschlossen sind, und danach ein AHB, an dem ausschließlich nicht Split-Capable Slaves angeschlossen sind. Somit genügt es, in einem „Durchlauf“ nur einen Satz Busphasen für einen Slave zu definieren. Dieser Satz kann dann unter Umbenennung der slavespezifischen Signale für jeden Slave des aktuell betrachteten AHB angenommen werden. Im Folgenden wird die Bestimmung der Busphasen und das Mapping für Split-Capable Slaves beschrieben, die Betrachtungen für den AHB ohne Split-Capable Slaves erfolgt im Abschnitt I.7.

Die Phasen werden nun gemäß Definition 3.7 festgelegt. Die Portzuordnung $pz(s)$ für den exemplarischen Slave s ergibt sich in trivialer Weise aus den in Abbildung I.1 gezeigten Signalen des Slave-Interfaces, da Portname, Bitbreite und Richtung dargestellt sind. Ein Port aus dieser Portzuordnung ist per Definition 3.1 z.B. $(\text{"}HSEL\text{"}, 1, a)$. Zur Verkürzung der Folgenden Notationen, sei für ein solches Tupel stellvertretend nur der Portname (im Beispiel also nur HSEL) genannt, da über diesen eine eindeutige Zuordnung zum eigentlichen Tupel möglich ist.

Port	HMASTER	HMASTLOCK	HSPLIT	HSEL	HTRANS	HADDR	HWRITE
Default	0	0	0	0	IDLE	0	0

Fortsetzung						
Port	HSIZE	HBURST	HPROT	HWDATA	HRDATA	HRESP
Default	0	0	0	0	0	0

Fortsetzung		
Port	HREADYOUT	HREADYIN
Default	1	0

Tabelle I.4.: Defaultzustände für die Ports im AHB-Slave-Interface

Tabelle I.4 definiert die Defaultzustände der Ports des Slave-Interfaces. Sind alle Signale im Defaultzustand so finden im Slave-Interface keine Transfers statt, da der Slave nicht adressiert (HSEL=0) ist.

Tabelle I.5 zeigt die Phasenports, Beobachtungsdauern und Funktionen der Busphasen im Slave-Interface und benennt ihre Typen gemäß Anhang B. Die Bedeutungen der Phasen werden im Folgenden erläutert und danach dann die Start-, Ende und Abbruchkriterien bestimmt und gegebenenfalls werden Besonderheiten hervorgehoben. Zur Straffung der Notation wird ähnlich wie im Master-Interface ein Hilfskonstrukt benötigt:

$$LRO = \{ \quad ((w_1, ..., w_{max+1}), n) \in (SI_{HREADYOUT})^{max+1} \times \mathbb{N} \mid \\ n \leq max + 1 \wedge w_n = 1 \wedge (\forall i \in \mathbb{N}, n < i \leq max + 1 : w_i = 0)\}$$

Ein Paar (w, n), bestehend aus einem $max + 1$ Takte langen Signalablauf von HREADYOUT w und einem Index n dieses Teilabschnittes, ist genau dann in LRO (LRO steht für „letztes ReadyOut“), wenn in Takt n HREADYOUT das letzte Mal auf Eins und danach nur noch auf Null war. In jedem AHB-spezifikationskonformen System kann zu jedem beliebigen $max + 1$ Takte lange Teilabschnitt w des gesamten Signalablaufs von HREADYOUT genau ein n gefunden werden, sodass $(w, n) \in LRO$ gilt, da wie oben beschrieben HREADY maximal max Takte lang Null sein darf.

SPLIT Benötigt ein Slave länger als max Takte, um einen Datenzyklus abzuschließen, würde er dadurch die durch AHB-Spezifikation vorgeschriebene maximale Anzahl Takte, während denen HREADYOUT Null sein darf, überschreiten. In diesem Fall kann ein Slave das Setzen von HRESP=SPLIT als Zweitaktantwort auf einen Datenzyklus den Master und den Arbiter informieren, dass er die max Takte Antwortverzögerung nicht einhalten kann. Daraufhin wird der Transfer abgebrochen und der Master muss es erneut versuchen. Zusätzlich wird der Master von der Arbitrierung ausgeschlossen, damit nicht sofort wieder die gleiche Situation entsteht. Sobald der Slave dann in der Lage ist, den Datenzyklus zu beenden, setzt er für genau einen Takt auf seinem HSPLIT-Ausgang das Bit, das dem von der Arbitrierung ausgeschlossenen Masters zugeordnet ist. Daraufhin wird der Master wieder in die Arbitrierung aufgenommen und kann dann den Transfer mit dem Slave erneut versuchen. Es bietet sich also ein Einzeltaktphase zur Modellierung an.

	PP_Φ				
Phasenname	$PP_{\Phi S}$	$PP_{\Phi E}$	$PP_{\Phi O}$	tb_Φ	Typ
SPLIT	{HSPLIT}	$\emptyset$	$\emptyset$	2	Einzeltakt
REQSL	{HSEL, HMASTER, HMAST-LOCK, HTRANS, HADDR, HWRITE, HSIZE, HBURST, HPROT, HREADYIN}	$\emptyset$	$\emptyset$	2	Einzeltakt
BURSTREQSL	{HSEL, HMASTER, HMAST-LOCK, HTRANS, HADDR, HWRITE, HSIZE, HBURST, HPROT}	$\emptyset$	{HREADY-OUT, HREA-DYIN}	2	Multitakt
DATASL	{HWDATA, HREADY-OUT}	{HREADY-OUT}	{HTRANS, HWRITE, HREADYIN}	$max + 2$	Multitakt
RESPSL	{HRDATA, HREADY-OUT}	$\emptyset$	{HTRANS, HWRITE, HREADYIN}	$max + 2$	Einzeltakt
NOTOKSL	{HRESP, HREADY-OUT}	$\emptyset$	{HTRANS, HREADYIN}	$max + 2$	Multitakt

Tabelle I.5.: Phasenstartports, Phasenendports, Beobachtungsdauern und Phasentypen im AHB-Slave-Interface

$S_{SPLIT} = \{(w_1, w_2) \in (SI_{HSPLIT})^2 | w_1 = 0 \wedge w_2 \neq 0\}$

$E_{SPLIT} = S_{SPLIT}$

$A_{SPLIT} = \emptyset,$

REQSL Die Phase REQSL entspricht einem NONSEQ-Addresszyklus, der an den Slave gerichtet ist (HSEL=1). Sie unterscheidet sich von den SEQ- oder BUSY-Addresszyklen dadurch, dass der Slave grundsätzlich keine Flusskontrolle über diese Phase hat, da die Gültigkeit über das globale HREADY (HREADYIN) gesteuert wird, welches zu diesem Zeitpunkt von dem Slave getrieben wird, welcher aktuell im Datenzyklus ist. Dies ist nicht notwendigerweise der Slave, an den gerade der NONSEQ-Addresszyklus gerichtet ist. Jedoch benötigt der Slave die Information, wann der Datenzyklus des potentiell anderen Slaves endet, damit er im darauffolgenden Takt selbst in den NONSEQ-Datenzyklus gehen kann. Nachdem also ein solcher NONSEQ-Addresszyklus startet, muss später nocheinmal eine Information über das Ende des Zyklus an den Slave übergeben werden. Man könnte also die REQSL-Phase mit dem Beginn eines NONSEQ-Addresszyklus starten lassen und diese dann, wenn HREADYIN=1 ist, abbrechen, sodass der Slave weiß, dass nun die Datenphase starten kann. Dies ist eher unnatürlich und würde bei einem dauerhaft auf Eins liegenden HREADYIN dazu führen, dass die REQ-Phase immer in einem Takt startet und stets im selben Takt abbricht. Darüber hinaus darf ein Slave im AHB die Signale des Addressbus nur bei HREADYIN=1 auswerten, die Information über den Start eines NONSEQ-Addresszyklus während HREADYIN=0, trägt für den Slave also keine verwertbare Information. Aus diesem Grund wird die REQSL-Phase als Einzeltaktphase definiert, die nur den Takt markiert, in dem HREADYIN=1 ist und ein NONSEQ-Addresszyklus and den Slave vorliegt.

$$S_{REQSL} = \{ \quad (w_1, w_2) \in \left(\bigtimes_{p \in PP_{REQSL}} SI_p \right)^2 \mid$$
$$proj_{HTRANS}(w_2) = NONSEQ \wedge proj_{HSEL}(w_2) = 1 \wedge$$
$$proj_{HREADYIN}(w_2) = 1\}$$

$E_{REQSL} = S_{REQ}$

$A_{REQSL} = \emptyset$

BURSTREQSL Die Phase BURSTREQSL entspricht SEQ- und BUSY-Addresszyklen. Für diese Zyklen hat der Slave die Flußkontrolle, da er sich gleichzeitig im Datenzyklus befindet, also sein HREADYOUT dem globalen HREADY entspricht. Während dieser Phasen erfordert das AHB-Protokoll, dass die Signale des Addressbusses den durch den aktuellen Burst vorgeschriebenen Abläufen entsprechen. Hier kann also keine Einzeltaktphase verwendet werden, die nur dann aktiv ist, wenn die Signale auch ausgewertet werden dürfen (bei HREADYOUT=1). Wäre dem so, würden während HREADYOUT=0 der Addressbus auf die Defaultwerte zurückfallen und so diesen Vorgaben widersprechen. Somit wird für diese Addresszyklen eine Multitaktphase verwendet.

$$
\begin{aligned}
S_{BURSTREQSL} = \{ \quad & (w_1, w_2) \in \left(\bigtimes_{p \in PP_{BURSTREQSL}} SI_p \right)^2 \mid \\
& [(proj_{HTRANS}(w_1) = NONSEQ \vee proj_{HTRANS}(w_1) = SEQ) \wedge \\
& \quad proj_{HTRANS}(w_2) = BUSY \wedge proj_{HSEL}(w_2) = 1] \\
& \vee \\
& [proj_{HREADYOUT}(w_1) = 1 \wedge proj_{HSEL}(w_2) = 1 \wedge \\
& \quad proj_{HTRANS}(w_1) = SEQ \wedge proj_{HTRANS}(w_2) = SEQ] \\
& \vee \\
& [proj_{HTRANS}(w_1) \neq SEQ \wedge \\
& \quad proj_{HTRANS}(w_2) = SEQ \wedge proj_{HSEL}(w_2) = 1]\}
\end{aligned}
$$

Die Phase startet, wenn der Slave ausgewählt ist und HTRANS von NONSEQ oder SEQ auf BUSY wechselt oder wenn im vorangegangenen Takt ein SEQ-Addresszyklus beendet wurde und gleich ein neuer startet oder wenn im vorangegangenen Takt kein SEQ-Addresszyklus (also NONSEQ oder BUSY) stattfand und jetzt ein SEQ-Addresszyklus folgt.

$$
\begin{aligned}
E_{BURSTREQSL} = \{ \quad & (w_1, w_2) \in \left(\bigtimes_{p \in PP_{BURSTREQSL}} SI_p \right)^2 \mid \\
& proj_{HTRANS}(w_2) = SEQ \wedge proj_{HREADYOUT}(w_2) = 1 \wedge \\
& proj_{HSEL}(w_2) = 1\}
\end{aligned}
$$

$$
\begin{aligned}
A_{BURSTREQSL} = \{ \quad & (w_1, w_2) \in \left(\bigtimes_{p \in PP_{BURSTREQSL}} SI_p \right)^2 \mid \\
& [proj_{HTRANS}(w_2) = IDLE \vee proj_{HTRANS}(w_2) = NONSEQ] \wedge \\
& proj_{HSEL}(w_1) = 1 \wedge \\
& [proj_{HTRANS}(w_1) = BUSY \vee \\
& proj_{HREADYOUT}(w_1) = 0 \wedge proj_{HTRANS}(w_1) = SEQ]\}
\end{aligned}
$$

Man beachte, dass eine BURSTREQSL-Phase nur im Falle eines SEQ-Addresszyklus endet. Eine BURSTREQSL-Phase, die einen (oder mehr) BUSY-Addresszyklus modelliert, endet nie. Sie kann lediglich abbrechen oder sie wird durch einen Neustart der BURSTREQSL-Phase (dann als SEQ-Addresszyklus) ersetzt. HREADYOUT gehört zu den Beobachtungsports der Phase, da sich HREADYOUT während HTRANS auf BUSY ist (also während die Phase gestartet aber noch nicht beendet ist) ändern kann. Der Zustand von HREADYOUT während der BURSTREQ-Phase wird durch den stets parallel dazu laufenden Datenzyklus bestimmt.

DATASL Die DATASL-Phase entspricht einem NONSEQ- oder SEQ-Schreibdatenzyklus. Sie beginnt, sobald ein NONSEQ- oder SEQ-Schreib-Addresszyklus beendet ist (und somit die Schreibdaten gültig werden) und endet, wenn der Slave die Datenübernahme mit HREADYOUT quittiert hat. Offenbar wird der Datenzyklus von Signalen, die nicht vom Datenzyklus steuerbar sind, kontrolliert. Dementsprechend werden diese Signale als Observierungsports verwendet.

$$S_{DATASL} = \{ \quad (w_1, w_2) \in \left(\bigtimes_{p \in PP_{DATASL} \cup PP_{DATASLO}} SI_p \right)^2 \mid$$
$$[(proj_{HTRANS}(w_1) = NONSEQ \wedge proj_{HREADYIN}(w_1) = 1)$$
$$\vee$$
$$(proj_{HTRANS}(w_1) = SEQ \wedge proj_{HREADYOUT}(w_1) = 1)] \wedge$$
$$proj_{HSEL}(w_1) = 1 \wedge proj_{HWRITE}(w_1) = 1\}$$

$$E_{DATASL} = \{ \quad (w_1, ..., w_{max+2}) \in \left(\bigtimes_{p \in PP_{DATASL} \cup PP_{DATASLO}} SI_p \right)^{max+2} \mid$$
$$proj_{HREADYOUT}(w_{max+2}) = 1 \wedge$$
$$([\exists n \in \mathbb{N}, 1 \leq n \leq max + 1 :$$
$$(proj_{HWRITE}(w_n) = 1 \wedge proj_{HTRANS}(w_n) = NONSEQ \wedge$$
$$proj_{HSEL}(w_n) = 1 \wedge$$
$$((proj_{HREADYIN}(w_\nu))_{\nu=1...max+1}, n) \in LRO)]$$
$$\vee$$
$$[\exists n \in \mathbb{N}, 1 \leq n \leq max + 1 :$$
$$(proj_{HWRITE}(w_n) = 1 \wedge proj_{HTRANS}(w_n) = SEQ \wedge proj_{HSEL}(w_n) = 1 \wedge$$
$$((proj_{HREADYOUT}(w_\nu))_{\nu=1...max+1}, n) \in LRO)])\}$$

Die Phase endet, wenn ein HREADYOUT gesetzt wird und beim letzten HREADY ein SEQ-oder NONSEQ-Addresszyklus beendet wurde und HREADYOUT von diesem letzten HREADY bis zum aktuellen Takt Null war. Dieses letzte HREADY ist HREADYIN, wenn es ein NONSEQ-Zyklus war und HREADYOUT, wenn es ein SEQ-Zyklus war. Da $SI_{HREADYOUT} = SI_{HREADYIN}$ gilt, kann LRO auch für die Überprüfung von HREADYIN genutzt werden.

$A_{DATA} = \emptyset$

RESPSL Diese Einzeltaktphase markiert den Takt, in dem ein NONSEQ- oder SEQ-Lesedatenzyklus endet. Der Beginn eines solchen Datenzyklus muss nicht markiert werden, da dabei (im Gegensatz zum Schreib-Datenzyklus) keine Informationen übertragen werden.

$$S_{RESPSL} = \{ \quad (w_1, ..., w_{max+2}) \in \left(\bigtimes_{p \in PP_{RESPSL} \cup PP_{RESPSLO}} SI_p \right)^{max+2} \mid$$
$$proj_{HREADYOUT}(w_{max+2}) = 1 \wedge$$
$$([\exists n \in \mathbb{N}, 1 \leq n \leq max + 1 :$$
$$(proj_{HWRITE}(w_n) = 0 \wedge proj_{HTRANS}(w_n) = NONSEQ \wedge$$
$$proj_{HSEL}(w_n) = 1 \wedge$$
$$((proj_{HREADYIN}(w_\nu))_{\nu=1...max+1}, n) \in LRO)]$$
$$\vee$$
$$[\exists n \in \mathbb{N}, 1 \leq n \leq max + 1 :$$
$$(proj_{HWRITE}(w_n) = 0 \wedge proj_{HTRANS}(w_n) = SEQ \wedge proj_{HSEL}(w_n) = 1 \wedge$$
$$((proj_{HREADYOUT}(w_\nu))_{\nu=1...max+1}, n) \in LRO)])\}$$

$E_{RESPSL} = S_{RESPSL}$

$A_{RESPSL} = \emptyset$

NOTOKSL (lies „not Okay Slave“) Per AHB-Protokoll haben Datenzyklen, die nicht mit HRESP=OK enden eine Zweitakt-Antwort. Im ersten Takt der Antwort wechselt HRESP auf einen Wert ungleich OK, wobei HREADY Null sein muss und im zweiten Takt muss dann HREADY auf Eins gehen, um den Datenzyklus abzuschliessen. Dementsprechend startet die NOTOKSL-Phase im ersten Takt dieser Zweitakt-Antwort und endet im darauffolgenden Takt mit HREADY auf Eins.

$$
\begin{aligned}
S_{NOTOKSL} = \{ \quad & (w_1, ..., w_{max+2}) \in \left(\bigtimes_{p \in PP_{NOTOKSL} \cup PP_{NOTOKSLO}} SI_p \right)^{max+2} \quad | \\
& proj_{HRESP}(w_{max+2}) \neq OK \wedge \\
& proj_{HREADYOUT}(w_{max+2}) = 0 \wedge \\
& ([\exists n \in \mathbb{N}, 2 \leq n \leq max + 1 : \\
& (proj_{HTRANS}(w_n) = NONSEQ \wedge proj_{HSEL}(w_n) = 1 \wedge \\
& ((proj_{HREADYIN}(w_\nu))_{\nu=1...max+1}, n) \in LRO)] \\
& \vee \\
& [\exists n \in \mathbb{N}, 2 \leq n \leq max + 1 : \\
& (proj_{HTRANS}(w_n) = SEQ \wedge proj_{HSEL}(w_n) = 1 \wedge \\
& ((proj_{HREADYOUT}(w_\nu))_{\nu=1...max+1}, n) \in LRO)])\}
\end{aligned}
$$

Man beachte, dass n größer oder gleich Zwei sein muss. Der Grund dafür ist, dass der erste Takt der Zweitakt-Antwort HREADY=0 hat, also zu der maximal zulässigen Verzögerung hinzuzählt. Somit kann das letzte HREADY nur noch max, anstelle von $max + 1$ Takten in der Vergangenheit liegen.

$E_{NOTOKSL} = \emptyset$

$$
\begin{aligned}
A_{NOTOKSL} = \{ \quad & (w_1, ..., w_{max+2}) \in \left(\bigtimes_{p \in PP_{NOTOKSL} \cup PP_{NOTOKSLO}} SI_p \right)^{max+2} \quad | \\
& proj_{HRESP}(w_{max+2}) \neq OK \wedge proj_{HREADYOUT}(w_{max+2}) = 1 \wedge \\
& proj_{HREADYOUT}(w_{max+1}) = 0 \wedge proj_{HRESP}(w_{max+1}) \neq OK\}
\end{aligned}
$$

Bei genauer Betrachtung der Definitionen der Defaultwerte und Phasen im Slave-Interface erkennt man, dass HREADYIN immer im Defaultzustand ist, außer wenn die Einzeltaktphase REQSL startet (und gleichzeitig endet). In allen anderen Phasen gehört HREADYIN nie zu den Phasenstart- oder -endports und sein Wert wird dementsprechend nicht vom Default abweichend festgelegt. Dies ist akzeptabel, da HREADYIN, während der Slave nicht in einem SEQ- oder NONSEQ-Datenzyklus ist, also wenn nicht kommuniziert wird, ohne Bedeutung für den Slave ist. Einzige Ausnahme ist ein NONSEQ-Addresszyklus, jedoch wird für diesen Fall ja HREADYIN bestimmt. Innerhalb von NONSEQ- oder SEQ-Datenzyklen genügt es HREADYOUT zu betrachten, da dann HREADYIN und HREADYOUT per AHB-Protokoll gleich laufen. Im *J-R*-Modell wird aber HREADYIN zu diesen Zeiten stets Null sein. Der

Gleichlauf wird also nicht explizit modelliert, denn ansonsten müsste eine weitere Phase zum Setzen von HREADYIN definiert werden, die immer genau dann startet, wenn der Slave die DATASL-Phase beendet oder die RESPSL-Phase startet (also HREADYOUT auf Eins setzt). Diese Information ist aber, wie oben beschrieben, bei einem AHB-protokollkonformen Kommunikationsautomaten redundant und wird deswegen nicht erfasst.

Damit sind alle Phasen des Slaves erfasst und es kann nun das GP- und TLM-Phase-Mapping durchgeführt werden.

I.4. GP- und TLM-Phase-Mapping: Master

Auf die oben bestimmten Busphasen für AHB-Master und -Slaves wird nun das in Abschnitt 4.3 beschriebene GP- und dann das TLM-Phase-Mapping angewendet.

Für die Phase BUSREQ des AHB-Masters legt das Startkriterium den Wert des Ports HBUSREQ beim Start auf Eins fest, während das Abbruchkriterium den Wert auf Null festlegt. Ein Endkriterium ist nicht vorhanden. Somit trifft für den Port HBUSREQ die in Schritt 1.1 der GP-Mappings festgelegte Bedingung zu und es muss kein GP-Element für HBUSREQ verwendet werden. Da es neben HBUSREQ keine anderen Ports in der Phase BUSREQ gibt, ist das GP für diese Phase ohne Bedeutung. Die Information der Phase, also der Zustand von HBUSREQ wird einzig über das Auftreten von Start oder Abbruch der Phase bestimmt. Gleiches gilt für die Phasen LOCK mit ihrem einzigen Port HLOCK und für die Phase GRANT mit ihrem einzigen Port HGRANT zu.

Für die Phase REQ müssen mehr als nur ein Port untersucht werden. Der Wert HTRANS wird in jedem Kriterium auf genau einen Wert festgelegt und muss somit nicht auf das GP abgebildet werden (Regel 1.1). HADDR kann auf das GP-Grundelement `m_address` abgebildet werden (Regel 1.3). HWRITE bestimmt, ob es sich um ein Schreib- oder einen Lesezugriff handelt, wird also auf das GP-Grundelement `m_command` abgebildet. HSIZE, HBURST und HPROT werden auf GP-Erweiterungen abgebildet, die den als Zahl interpretierten Wert der Signale speichern können (Regel 1.4). Für alle wird `short` als Datentyp gewählt. HREADY wird durch das Ende der Phase auf Eins festgelegt. Daraus folgt, dass wenn die Phase gestartet ist und nicht endet, HREADY Null sein muss (sonst würde sie enden). Dieser Tatsache ist mit Regel 1.2 im GP-Mapping Sorge getragen, sodass kein Wert im GP für HREADY nötig wird.

Für die Phase BURSTREQ ergibt sich abgesehen von HTRANS und HREADY das gleiche Mapping wie für die Phase REQ. HTRANS wird bei BURSTREQ nicht durch alle Kriterien festgelegt und somit wird eine GP-Erweiterung definiert, deren Wert die zwei bei BURSTREQ möglichen Werte speichern kann (Regel 1.4). Um die Arbeit mit der Erweiterung zu vereinfachen, wird nicht einfach eine Zahl (0 oder 1) gespeichert, sondern ein C++-Aufzählungstyp, der die Werte `BUSY und SEQ` umfasst.

Bei der Phase DATA wird HWDATA auf das Datenfeld des GP abgebildet. HREADY wird nur beim Phasenende festgelegt, aber Regel 1.2 erlaubt es, wie oben beschrieben, auf

eine Abbildung auf ein GP-Element zu verzichten. Bei der Phase RESP wird HRDATA auch auf das Datenfeld abgebildet, da die RESP- und die DATA-Phasen nie gleichzeitig bei einer Transaktion aktiv sein können. Das Startkriterium (und somit auch das Endkriterium) der Einzeltaktphase RESP legt HREADY auf Eins fest, sodass mittels Regel 1.1 auch bei dieser Phase von einem GP-Element für HREADY abgesehen werden kann.

Bei der Phase NOTOK wird der ERROR-Wert von HRESP auf das Response-Status-Element des GP und dessen von `TLM_[OK|INCOMPLETE]_RESPONSE` abweichende Werte abgebildet. Dies erlaubt in der Simulation zusätzlich noch den eigentlichen Grund des Errors (z.B. Address- oder Command- oder Burst-Error) mitzuteilen und so das Analysieren des Systems zu vereinfachen. Die Werte RETRY und SPLIT werden auf eine GP-Erweiterung abgebildet, die einen Boolschen Wert trägt, der bei `true` für SPLIT und bei `false` für Retry steht, jedoch nur wenn die GP-Erweiterung gültig ist. Der HREADY-Port der NOTOK-Phase muss gemäß Regel 1.1 nicht im GP enthalten sein.

Tabelle I.6 zeigt nochmals das Ergebnis des Mappings.

Port	Abbildung
HBUSREQ	Implizit über Start, Abbruch und Abwesenheit der BUSREQ-Phase
HLOCK	Implizit über Start, Abbruch und Abwesenheit der LOCK-Phase
HGRANT	Implizit über Start, Abbruch und Abwesenheit der GRANT-Phase
HTRANS	Implizit über Start, Abbruch und Abwesenheit der REQ-Phase und explizit als GP-Erweiterung `transfer_type` mit Typ `enum`
HADDR	Explizit als GP-Grundelement `m_address`
HWRITE	Explizit als GP-Grundelement `m_command`
HSIZE	Explizit als GP-Erweiterung `transfer_size` mit Typ `short`
HBURST	Explizit als GP-Erweiterung `burst_type` mit Typ `short`
HPROT	Explizit als GP-Erweiterung `prot_info` mit Typ `short`
HWDATA	Explizit als GP-Grundelemente `m_data[_length]`
HRDATA	Explizit als GP-Grundelemente `m_data[_[length]`
HRESP	Explizit als GP-Grundelement `m_response_status` und eine GP-Erweiterung `splitNrety` mit Typ `bool`
HREADY	Implizit über Starts, Enden, Abbrüche diverser Phasen

Tabelle I.6.: GP-Mapping-Ergebnis für die Busphasen im AHB-Master-Interface

Nach dem GP- folgt nun das TLM-Phase-Mapping für die Phasen des Masters. Bei Beginn des Mappings ist noch keine TLM-Phase bestimmt und somit wird die in Schritt 1 des TLM-Phase-Mappings beschriebene Suche erfolglos sein und Schritt 2 führt zur Definition von `BEGIN_BUSREQ` deren definierende Busphase dann BUSREQ ist. Schritte 3 und 4 müssen aufgrund der Abwesenheit eines Endkriteriums nicht durchgeführt werden. Die Suche in Schritt 5 wird erfolglos sein, da die einzige bisher definierte TLM-Phase das Präfix

BEGIN_ trägt. Somit wird in Schritt 6 eine neue TLM-Phase ABORT_BUSREQ definiert, deren definierende Busphase auch BUSREQ ist.

Beim Mapping von LOCK wird in Schritt 1 BEGIN_BUSREQ ein potentieller Kandidat für das Mapping sein, da diese TLM-Phase das richtige Präfix trägt. Jedoch wird die Bedingung 1.1 verletzt, da die mittels HBUSREQ und HLOCK übertragenen Informationen nicht sinnvoll als gleich angesehen werden können. Schritt 2 wird also die TLM-Phase BEGIN_LOCK definieren. Schritt 3 und 4 finden aufgrund der Abwesenheit des Endkriteriums wiederum keine Anwendung. Schritte 5 und 6 werden zur Definition von ABORT_LOCK führen.

Für die Phase GRANT gilt das gleiche wie für die Phase LOCK und es werden die TLM-Phasen BEGIN_GRANT und ABORT_GRANT definiert.

Beim TLM-Phase-Mapping der Phase REQ entstehen die TLM-Phasen BEGIN_REQ, END_REQ und ABORT_REQ, da bei dieser Phase ausschließlich bisher nicht betrachtete Ports verwendet werden. Für die Phase BURSTREQ entsteht die TLM-Phase BEGIN_BURSTREQ, da keine der bereits existierenden TLM-Phasen mit BEGIN_-Präfix die Kriterien 1.1 bis 1.3 erfüllen: Die Phasen BEGIN_BUSREQ, BEGIN_LOCK und BEGIN_GRANT scheitern am Kriterium 1.2, da sie keinerlei GP-Elemente nutzen, während BURSTREQ durchaus GP-Elemente nutzt. Die Phase BEGIN_REQ scheitert an Kriterium 1.1, da bei Phase REQ HTRANS implizit modelliert wird, während BURSTREQ dafür ein GP-Element nutzt. Folglich führen die Schritte 3 bis 6 dann zu Bestimmung von END_BURSTREQ und ABORT_BURSTREQ.

Für die Busphase DATA entsteht die Phase BEGIN_DATA, da HWDATA auf ein von den anderen Busphasen ungenutztes GP-Grundelement abgebildet wird und somit entseht auch END_DATA.

Bei der Busphase RESP entstehen die TLM-Phasen BEGIN_RESP und END_RESP. Die naheliegendste, bisher definierte TLM-Phase für BEGIN_RESP wäre BEGIN_DATA und sie stimmt auch in der Verwendung der GP-Elemente (m_data) und der implizit modellierten Ports (HREADY=1) überein, jedoch ist die mit HREADY übertragene Information (Lesedaten sind gültig) eine andere als bei der DATA-Phase (Schreibdaten wurden übernommen). Schritt 3 folgend entsteht dann auch END_RESP.

Für die Phase NOTOK entstehen die TLM-Phasen BEGIN_NOTOK und ABORT_NOTOK, da nur NOTOK den Port HRESP und dessen GP-Mapping verwendet.

I.5. GP- und TLM-Phase-Mapping: Slave

Für die Phase REQSL müssen mehrere Ports untersucht werden. Die Werte von HTRANS, HREADYIN und HSEL werden im Startkriterium (und da es sich um eine Einzeltaktphase handelt also auch im Endkriterium) auf genau einen Wert festgelegt und müssen somit nicht auf das GP abgebildet werden (Regel 1.1). HADDR kann auf das GP-Grundelement m_address abgebildet werden (Regel 1.3). HWRITE bestimmt, ob es sich um ein Schreib- oder einen Lesezugriff handelt, wird also auf das GP-Grundelement m_command abgebildet. HSIZE, HBURST und HPROT werden auf die gleichen GP-Erweiterungen abgebildet, auf

die die entsprechenden Ports im Master-Interface abgebildet wurden, da sie im Kontext der REQSL-Phase die gleiche Information übertragen wie beim Master in der REQ-Phase (Regel 1.3). Die Ports HMASTER und HMASTLOCK werden auf zusätzliche Erweiterungen mit Typ `int` bzw. `bool` abgebildet, damit während dieser Phasen die ID des aktiven Masters und die Information, ob die Transaktion zu einem größeren Verbund von Transaktionen („Locked Transactions“) gehört, übertragen werden kann.

Port	Abbildung
HSPLIT	Explizit als GP-Erweiterung `split_mask` mit Typ `int`
HMASTER	Explizit als GP-Erweiterung `master_id` mit Typ `int`
HMASTLOCK	Explizit als GP-Erweiterung `is_locked` mit Typ `bool`
HSEL	Implizit über Starts, Enden, Abbrüche von REQSL, BURSTREQSL
HTRANS	Implizit über Start, Abbruch und Abwesenheit der REQSL-Phase und explizit als GP-Erweiterung `transfer_type` mit Typ `enum`
HADDR	Explizit als GP-Grundelement `m_address`
HWRITE	Explizit als GP-Grundelement `m_command`
HSIZE	Explizit als GP-Erweiterung `transfer_size` mit Typ `short`
HBURST	Explizit als GP-Erweiterung `burst_type` mit Typ `short`
HPROT	Explizit als GP-Erweiterung `prot_info` mit Typ `short`
HWDATA	Explizit als GP-Grundelemente `m_data[_length]`
HRDATA	Explizit als GP-Grundelemente `m_data[_length]`
HRESP	Explizit als GP-Grundelement `m_response_status` und eine GP-Erweiterung `splitNretry` mit Typ `bool`
HREADYIN	Implizit über Start und Ende der REQSL-Phase
HREADYOUT	Implizit über Starts, Ende und Abbrüche diverser Phasen

Tabelle I.7.: GP-Mapping-Ergebnis für die Busphasen im AHB-Slave-Interface

Für die Phase BURSTREQSL ergibt sich abgesehen von HTRANS und HREADYIN das gleiche Mapping wie für die Phase REQSL. HTRANS wird bei BURSTREQ nicht durch alle Kriterien festgelegt und wird auf die gleiche Erweiterung abgebildet, die auch in der BURSTREQ-Phase im Master-Interface die Information von HTRANS trug. HREADYIN wird von BURSTREQSL nicht verwendet.

Bei der Phase DATASL wird HWDATA auf das Datenfeld des GP abgebildet. HREADYOUT wird nur beim Phasenende festgelegt, aber Regel 1.2 erlaubt es, wie oben beschrieben, auf eine Abbildung auf ein GP-Element zu verzichten. Bei der Phase RESPSL wird HRDATA auch auf das Datenfeld abgebildet, da die RESPSL- und die DATASL-Phasen nie gleichzeitig bei einer Transaktion aktiv sein können. Das Startkriterium (und somit das Endkriterium) der Einzeltaktphase RESPSL legt HREADYOUT auf Eins fest, sodass mittels Regel 1.1 auch bei dieser Phase von einem GP-Element für HREADYOUT abgesehen werden kann.

Bei der Phase NOTOKSL wird der ERROR-Wert von HRESP auf das Response-Status-Element des GP und dessen von `TLM_[OK|INCOMPLETE]_RESPONSE` abweichende Werte abgebildet. Die Werte RETRY und SPLIT werden auf die während des Mappings von HRESP in der NOTOK-Phase des Master-Interfaces definierte GP-Erweiterung abgebildet.

Beim GP-Mapping der Phase SPLIT wird zusätzlich noch eine Erweiterung mit Typ `int` zur Signalisierung der wieder zur Arbitrierung zugelassenen Master notwendig. Tabelle I.7 zeigt das Ergebnis des GP-Mappings im Slave-Interface.

Beim TLM-Phase-Mapping von SPLIT entstehen die Phasen `BEGIN_SPLIT` und `END_SPLIT`, da die Phase kein GP-Grundelement verwendet und auch kein Signal implizit modelliert, während alle bisher definierten TLM-Phasen dies tun.

Das Start- und Endkriterium von REQSL führt dazu, dass HREADYIN und HSEL implizit modelliert werden. Die Bedeutung von HREADYIN ist im Rahmen von REQSL „Die Werte des Addressbus dürfen gelesen werden“ und die von HSEL ist „Der Slave ist addressiert“. Keines der bisher implizit modellierten Signale deckt diese Bedeutungen ab und so müssen `BEGIN_REQSL` und `END_REQSL` zum Satz der TLM-Phasen hinzugefügt werden.

Beim TLM-Phase-Mapping von BURSTREQSL wird mit der bereits bei Mapping der Masterphasen definierten TLM-Phase `BEGIN_BURSTREQ` eine TLM-Phase gefunden, deren definierende Busphase (BURSTREQ aus dem Master-Interface) die gleichen GP-Elemente verwendet, jedoch nicht die gleichen Signale implizit modelliert, da BURSTREQSL HSEL implizit auf Eins festlegt. Alle anderen Phasen scheiden ebenso aus. Somit werden `[BEGIN|END|ABORT]_BURSTREQSL` zu den TLM-Phasen hinzugefügt.

Das TLM-Phase-Mapping von DATASL, RESPSL und NOTOKSL wird die bereits im TLM-Phase-Mapping des Master-Interfaces definierten TLM-Phasen `[BEGIN|END_DATA]`, `[BEGIN|END_RESP]` und `[BEGIN|ABORT_NOTOK]` verwenden, da die jeweiligen Phasen in den von ihnen verwendeten GP-Grundelementen und den implizit modellierten Signalen übereinstimmen (HREADY und HREADYOUT können im Kontext der jeweiligen Busphasen als gleich angesehen werden).

I.6. Zusammenfassung des TLM-Phase-Mappings

Tabelle I.8 zeigt das Ergebnis des TLM-Phase-Mappings für AHB-Master und Slave im Überblick. Man erkennt, dass sich alle TLM-Phasen, abgesehen von den `END_`-Phasen, in ihrer Kombination aus Präfix, implizit modellierten Signalen und verwendeten GP-Grundelementen gegenseitig unterscheiden, also alle Phasen unterschiedliche Information übertragen und so keine Redundanz existiert.

Die `END_`-Phasen haben immer das gleichen Suffix, wie die entsprechenden `BEGIN_`-Phasen, da dies eine intuitive Nutzung erlaubt. Aus diesem Grund kann es zu Redundanzen bei `END_`-Phasen kommen (zum Beispiel zwischen `END_SPLIT` und `END_RESP`). Diese Redundanz aufzulösen, führte aber zu nicht intuitiven und schwer vermittelbaren Abläufen wie beispielsweise `BEGIN_SPLIT` gefolgt von `END_RESP`.

TLM-Phase	definierende Phase	Repräsentant für	implizit modellierte Signale	GP-Grundelemente
BEGIN_LOCK	LOCK	Start: LOCK	HLOCK=1	keine
ABORT_LOCK	LOCK	Abbruch: LOCK	HLOCK=0	keine
BEGIN_BUSREQ	BUSREQ	Start: BUSREQ	HBUSREQ=1	keine
ABORT_BUSREQ	BUSREQ	Abbruch: BUSREQ	HBUSREQ=0	keine
BEGIN_GRANT	GRANT	Start: GRANT	HGRANT=1	keine
ABORT_GRANT	GRANT	Abbruch: GRANT	HGRANT=0	keine
BEGIN_REQ	REQ	Start: REQ	HTRANS=NONSEQ	m_command, m_address
ABORT_REQ	REQ	Abbruch: REQ	HTRANS=IDLE	m_command, m_address
END_REQ	REQ	Ende: REQ	HREADY=1	keine
BEGIN_BURSTREQ	BURSTREQ	Start: BURSTREQ	keine	m_command, m_address
ABORT_BURSTREQ	BURSTREQ	Abbruch: BURSTREQ	keine	m_command, m_address
END_BURSTREQ	BURSTREQ	Ende: BURSTREQ	keine	keine
BEGIN_DATA	DATA	Start: DATA, DATASL	HREADY(OUT)=0	m_data[_length]
END_DATA	DATA	Ende: DATA, DATASL	HREADY(OUT)=1	keine
BEGIN_RESP	RESP	Start: RESP, RESPSL	HREADY(OUT)=1	m_data[_length]
END_RESP	RESP	Ende: RESP, RESPSL	keine	keine
BEGIN_NOTOK	NOTOK	Start: NOTOK, NOTOKSL	HREADY(OUT)=0	m_response_status
ABORT_NOTOK	NOTOK	Abbruch: NOTOK, NOTOKSL	HREADY(OUT)=1	m_response_status
BEGIN_SPLIT	SPLIT	Start: SPLIT	keine	keine
END_SPLIT	SPLIT	Ende: SPLIT	keine	keine
BEGIN_REQSL	REQSL	Start: REQSL	HTRANS=NONSEQ, HSEL=1, HREADYIN=1	m_command, m_address
END_REQSL	REQSL	Ende: REQSL	keine	keine
BEGIN_BURSTREQSL	BURSTREQSL	Start: BURSTREQSL	HSEL=1	m_command, m_address
ABORT_BURSTREQSL	BURSTREQSL	Abbruch: BURSTREQSL	HSEL=1	m_command, m_address
END_BURSTREQSL	BURSTREQSL	Ende: BURSTREQSL	keine	keine

Tabelle I.8.: Übersicht über das Ergebnis des TLM-Phase-Mappings für AHB-Master und -Slave

I.7. Mehrfaches GP- und TLM-Phase-Mapping

Wie bereits weiter oben erwähnt muss der Prozess der Phasendefinition und des GP- und TLM-Phase-Mappings nochmals für einen AHB ohne Split-Capable Slaves wiederholt werden. Geht man dabei genauso vor wie oben beschrieben, gibt es nur Unterschiede in den Busphasen des Slaves. Die Busphase SPLIT, sowie die Ports HMASTLOCK und HMASTER aus den REQSL- und BURSTREQSL-Phasen entfallen. Somit werden die entsprechenden GP-Erweiterungen in den ersten drei Zeilen von Tabelle I.7 nicht beim GP-Mapping und die TLM-Phasen `BEGIN_SPLIT` und `END_SPLIT` nicht beim TLM-Phase-Mapping erzeugt.

I.8. Bestimmung der TLM-Phasenassoziation der GP-Erweiterungen

Für die im GP-Mapping als notwendig erkannten GP-Grundelemente und GP-Erweiterungen müssen nun die Phasenassoziationen bestimmt werden (siehe Abschnitt 4.7.1). Für die GP-Grundelemente kann dies sehr leicht aus Tabelle I.8 abgelesen werden und so ergeben sich die Phasenassoziationen der GP-Grundelemente wie im oberen Abschnitt der Tabelle I.9 gezeigt.

Die Phasenassoziationen der GP-Erweiterungen ergeben sich nach dem in Abschnitt 4.7.1 erläuterten Ablauf und es ergeben sich die Phasenassoziation wie im unteren Abschnitt von Tabelle I.9.

I.9. TLM-Phasenreduktion

Nach der Bestimmung der Phasenassoziationen kann nun eine TLM-Phasenreduktion (siehe Abschnitt 4.3.4) durchgeführt werden. Wie bereits im Abschnitt I.6 erläutert, gibt es grundsätzlich keine redundanten TLM-Phasen (abgesehen von den TLM-Phase, die das Ende von Phasen markieren). Dies gilt aber nur, wenn man nicht im Besitz der Information ist, ob die TLM-Phasen zwischen einem Master und einem AHB-Modell oder einem AHB-Modell und einem Slave ausgetauscht werden. Geht man aber davon aus, dass zwei kommunizierende Module wissen ob sie Master, Slave oder AHB-Modell sind und mit wem sie kommunizieren[3], kann die Liste der Phasen noch reduziert werden.

Unter dieser Annahme treten die TLM-Phasen `[BEGIN|ABORT|END]_REQ` und `[BEGIN|END|ABORT]_BURSTREQ` ausschließlich auf Verbindungen zwischen Master und AHB-Modell und die TLM-Phasen `[BEGIN|END]_REQSL` und `[BEGIN|END|ABORT]_BURSTREQSL` ausschließlich auf Verbindungen zwischen AHB-Modell und Slave auf.

Die mittels `[BEGIN|END]_REQ` und `[BEGIN|END]_REQSL` übertragene Informationen sind Start und Ende eines NONSEQ-Addresszyklus im Master- bzw. Slave-Interface. Die Informa-

[3]Eine durchaus sinnvolle Annahme.

GP-Element	assoziierte Phasen
GP-Grundelemente	
m_command	BEGIN_REQ, ABORT_REQ, BEGIN_BURSTREQ, ABORT_BURSTREQ, BEGIN_REQSL, BEGIN_BURSTREQSL, ABORT_BURSTREQSL
m_address	BEGIN_REQ, ABORT_REQ, BEGIN_BURSTREQ, ABORT_BURSTREQ, BEGIN_REQSL, BEGIN_BURSTREQSL, ABORT_BURSTREQSL
m_data[_length]	BEGIN_DATA, BEGIN_RESP
m_response_status	BEGIN_NOTOK, ABORT_NOTOK
GP-Erweiterungen	
transfer_type	BEGIN_BURSTREQ, ABORT_BURSTREQ, BEGIN_BURSTREQSL, ABORT_BURSTREQSL
transfer_size	BEGIN_REQ, ABORT_REQ, BEGIN_BURSTREQ, ABORT_BURSTREQ, BEGIN_REQSL, BEGIN_BURSTREQSL, ABORT_BURSTREQSL
burst_type	BEGIN_REQ, ABORT_REQ, BEGIN_BURSTREQ, ABORT_BURSTREQ, BEGIN_REQSL, BEGIN_BURSTREQSL, ABORT_BURSTREQSL
prot_info	BEGIN_REQ, ABORT_REQ, BEGIN_BURSTREQ, ABORT_BURSTREQ, BEGIN_REQSL, BEGIN_BURSTREQSL, ABORT_BURSTREQSL
master_id	BEGIN_REQSL, BEGIN_BURSTREQSL, ABORT_BURSTREQSL
is_locked	BEGIN_REQSL, BEGIN_BURSTREQSL, ABORT_BURSTREQSL
split_mask	BEGIN_SPLIT
splitNretry	BEGIN_NOTOK, ABORT_NOTOK

Tabelle I.9.: TLM-Phasenassoziation der GP-Elemente für den AHB

tionen sind vergleichbar. BEGIN_REQ und BEGIN_REQSL nutzen die gleichen GP-Grundelement und widersprechen sich nicht in den implizit modellierten Signalen. BEGIN_REQSL modelliert lediglich noch weitere Signale implizit. Es werden also alle Bedingungen für die TLM-Phasenreduktion erfüllt und BEGIN_REQ und BEGIN_REQSL können zu BEGIN_REQ zusammengelegt werden. Ob BEGIN_REQ den Start der REQ- oder REQSL-Phase bestimmt und welche Signale somit implizit modelliert werden, kann aus der Information, ob die Phase auf einer Master→AHB- oder AHB→Slave-Verbindung auftritt, ermittelt werden. Da der Start von REQSL nun mit BEGIN_REQ modelliert wird, wird für deren Ende dann auch END_REQ verwendet (Erklärung siehe Fußnote 7 auf Seite 44).

Entsprechend können auch BEGIN_BURSTREQ und BEGIN_BURSTREQSL zu BEGIN_BURSTREQ, ABORT_BURSTREQ und ABORT_BURSTREQSL zu ABORT_BURSTREQ und END_BURSTREQ und END_BURSTREQSL zu END_BURSTREQ zusammengelegt werden.

Damit werden [BEGIN|END]_REQSL und [BEGIN|END|ABORT]_BURSTREQSL in den Phasenassoziationen in Tabelle I.9 durch [BEGIN|END]_REQ bzw. [BEGIN|ABORT]_BURSTREQ ersetzt.

Entstehen dadurch Mehrfachnennungen in den Phasenassoziationen, können die Mehrfachnennungen durch einfache Nennungen ersetzt werden.

I.10. Änderungsintervalle der GP-Elemente

Die Festlegung, ob eine Transaktion ein Lese- oder Schreibzugriff ist, bleibt vom ersten Master bis zum finalen Slave unverändert und so wird `m_command` Ende-zu-Ende-invariant (e2e).

Im TLM-Modell sollen Bursts mit nur einem GP modelliert werden. Das bedeutet, dass das gleiche GP erstmals mit `BEGIN|END_REQ` (NONSEQ-Addresszyklus) übertragen wird und dann ggf. noch mehrfach in `BEGIN|END_BURSTREQ` übertragen wird (und natürlich auch entsprechend in der DATA- oder RESP-Phase). In jedem Addresszyklus wird auf dem AHB eine Adresse übertragen, jedoch folgt diese Adress fest vorgeschriebenen Abläufen (abhängig von HBURST und HSIZE), sodass ein Slave bereits nach dem Empfang des NONSEQ-Addresszyklus die Adresse aller folgenden SEQ-Addresszyklen kennt. Da der Anwendungsfall für das TLM-Modell Performanceanalyse sein soll, kann von fehlerhaften Addresssequenzen abstrahiert werden, da eine fehlerhafte Addresssequenz ein zwingend zu behebender Designfehler ist und in Systemen, deren Performance zu vermessen ist, nicht existieren darf. Somit wird `m_address` als Punkt-zu-Punkt-invariant (x2x) deklariert. So kann ein Master die NONSEQ-Adresse setzen und absenden. Danach darf er die Adresse nicht mehr ändern. Dies erlaubt dem Empfänger die korrekt Adresse aller folgenden SEQ-Addresszyklen zu bestimmen und andererseits auch eine einmalige Addressmodifikation bei der Weiterleitung des NONSEQ-Transfers durchzuführen. Dies bedeutet auch, dass das notwendige Setzen der Adressen beim Start der BURSTREQ-Phase (da HADDR zu den Phasenstartports gehört) nicht explizit modelliert wird. Der Wert von HADDR für SEQ-Adresszyklen kann aus den explizit modellierte Werten der NONSEQ-Phase ermittelt werden. Listing I.10 zeigt dies mit Hilfe einer C++-Funktion

Wie bereits erwähnt, soll ein Burst mit nur einem GP abgewickelt werden. Somit werden im Rahmen eines Bursts mehrere DATA- oder RESP-Phasen ablaufen. Am Ende eines Bursts soll das Datenfeld dann alle im Rahmen des Bursts übertragenen Datenelemente enthalten. Daher wird das Datenfeld als byteweise Ende-zu-Ende-invariant (siehe Abschnitt 4.7.7) festgelegt. Dazu wird die Phasenassoziation verfeinert: Jedes Byte wird mit einem speziellen Auftreten der Phasen aus der Phasenassoziation assoziiert. Vor diesem Auftreten muss das entsprechende Byte gesetzt werden und ist von da an Ende-zu-Ende-Invariant. Auf einer Punkt-zu-Punkt-Verbindung verfeinert sich die Phasenassoziation für das `m_data`-Feld, derart, dass Byte x des Datenfeldes mit dem n-ten Auftreten einer Phase aus der Phasenassoziation verbunden ist, wobei $n = \lfloor \frac{x}{size} \rfloor + 1$ gilt. Dabei ist $size$ der für den aktuellen Burst gültige Wert von 2^{HSIZE}. Die Allokation des Datenfeldes und somit auch die Festlegung der Länge des Feldes müssen vom Initiator zusammen mit der Erzeugung des GP stattfinden. Im Falle eines Bursts unbestimmter Länge kann der Master in der Regel eine Obergrenze

```
long long caluclate_next_address( //wird bei Empfang eines jeden BEGIN_REQ aufgerufen
  long long addr, //Die NONSEQ-Adresse
  int& n, //ein Zaehler, initial, also nach dem NONSEQ-Addresszyklus, auf Null
  int size, //Wert der transfer_size GP-Erweiterung beim NONSEQ-Zyklus
  int burst, //Wert der burst_type GP-Erweiterung beim NONSEQ-Zyklus
  is_SEQ //true, wenn transfer_type GP-Erweiterung ist SEQ, false bei BUSY
)
{
  int bytes_in_beat=((unsigned int)1)<<size;
  long long next_addr=addr+(n+1)* bytes_in_beat;
  if (!(burst & 0x1)){ //wrapping burst
    int exp=((burst&0x6)>>1)+1; //2,3 or 4
    int beats_in_burst= ((unsigned int)1)<<exp; //4,8 or 16
    unsigned int mask=(beats*bytes_in_beat)-1;
    long long wrap_start=addr & ~mask;
    long long wrap_boundary=wrap_start+mask;
    if (next_addr>wrap_boundary) next_addr-=(mask+1);
  }
  if (is_SEQ) n++; //nur bei echtem Transfer erhoehen
  return next_addr;
}
```

Listing I.10: Berechnung von HADDR bei BUSY oder SEQ Zyklen

für die Größe des Datenfeldes bestimmen (z.B. 1024 Transfers), da unendliche Bursts in der Realität nicht vorkommen.

Innerhalb eines AHB-Bursts können verschiedene HRESP-Werte erfolgen. Folgende beispielhafte Abfolge von Werten ist bei einem 8-Transfer-Burst legal: OK, OK, ERROR, OK, ERROR, RETRY. Nach einer ERROR-Response darf der Master entscheiden, ob er den Burst terminiert oder fortführt. Im Beispiel wurde er fortgeführt. Bei einer RETRY-Response muss der Master aber den Burst abbrechen (daher nur 6 Antworten). Offensichtlich ist die Response also in einer nicht vorhersehbaren Weise (im Gegensatz zur Adresse) zeitlich variabel. Gemäß Abschnitt 4.7.2 erhält sie also ein Punkt-zu-Punkt-variables (p2p) Änderungsintervall. Dieses Änderungsintervall trifft auch für die GP-Erweiterung `splitNretry` zu, da das Signalisieren von ERROR im obigen Ablauf bedeutet, die Erweiterung bei einer Phase ihrer Assoziation (`BEGIN_NOTOK`) als nicht gültig zu markieren, da ein ERROR weder SPLIT noch RETRY ist. Später aber wird sie als gültig markiert und gesetzt, um RETRY zu signalisieren. Somit muss auch für diese Erweiterung eine zeitliche Änderung zulässig sein.

Die GP-Erweiterung `transfer_type` kann innerhalb eines Bursts in vom Slave nicht vorhersehbarer Abfolge zwischen SEQ und BUSY zeitlich variieren. Somit muss diese GP-Erweiterung ein p2p-Änderungsintervall erhalten.

Die `transfer_size`, `burst_type`, `prot_info`, `master_id` und `is_locked` Erweiterungen sind für einen Burst zeitlich konstant, können aber aufgrund von Adaptionen oder Busbrücken, wie zum Beispiel Busbreitenkonvertierungen, zwischen zwei Punkt-zu-Punkt-Verbindungen variieren. Hier bietet sich das x2x-Änderungsintervall an. Das innerhalb der BURSTREQ-Phase notwendige Setzen dieser Signale geschieht ähnlich wie das der Adresse implizit als Wiederholen des Signales, das beim NONSEQ-Transfer aktiv war (der eigentli-

che Wert der Erweiterung könnte sich bereits wegen des x2x-Änderungsintervalls geändert haben).

Die Erweiterung `split_mask` erhält das e2e-Änderungsintervall, da eine Split-Information nur von einem Slave zu einem AHB-Modell, von dort aber nicht weiter gesendet werden kann. Der Transaktionspfad für eine Split-Information hat also immer die Länge Eins und kann sich so nicht räumlich ändern. Eine Transaktion mit Split-Information beginnt und endet mit `BEGIN_SPLIT` (siehe dazu auch den Unterabschnitt „Zusammenfassung“ dieses Abschnittes), sodass es auch keine zeitlichen Änderungen geben kann.

I.11. Implementierung der GP-Erweiterungen

Wie in Abschnitt 4.8.4 beschrieben, können die GP-Erweiterungen nach Belieben implementiert werden. Die Möglichkeit ihre Gültigkeit zu setzen und zu testen und ein performantes Memory-Management werden durch die in Abschnitt 4.8 gezeigten Mechanismen automatisch hinzugefügt. In diesem Beispiel entscheide ich mich für eine Implementierung, die für Pool-basiertes Memory-Management gedacht ist. Dies wird aber nur im Falle von Transaktionskopien Anwendung finden, da das Memory-Management der GP-Erweiterungen wie oben erwähnt automatisch verändert wird.

Es werden fünf Erweiterungen verschiedenen Typs benötigt. Es bietet sich die Verwendung einer in Anhang J, Listing J.2 gezeigten Template-Klasse und die in den Zeilen den Zeilen 35ff. von Listing J.2 gezeigten Implemtierungen der GP-Erweiterungen an.

I.12. Bindungschecks

Nachdem die zu verwendenden GP-Elemente, die notwendigen GP-Erweiterungen und deren Phasenassoziationen und Änderungsintervall bestimmt und implementiert sind, werden die Bindungchecks festgelegt. Wie in Abschnitt 4.6.1 erklärt, enthält der unverhandelbare Satz von GP-Erweiterungen nur die Erweiterungen, auf die beim GP-Mapping jedes Masters und jedes Slaves mindestens ein Port abgebildet wurde. Im AHB sind dies alle Erweiterungen, außer den Erweiterungen, die nur von einem Split-Capable Slave verwendet werden. Für die TLM-Phasen werden beim Mapping der Master die TLM-Phasen für Starts und Enden der Busphasen LOCK, BUSREQ und GRANT erzeugt bzw. verwendet, beim Mapping der Slaves aber nicht. Darüber hinaus werden die TLM-Phasen `[BEGIN|END]_SPLIT` nur beim Mapping von Split-Capable Slaves benutzt. Somit gehören diese TLM-Phasen nicht zum unverhandelbaren Satz. Tabelle I.11 zeigt den unverhandelbaren und den verhandelbaren Satz von GP-Erweiterungen bzw. TLM-Phasen nochmals im Überblick.

Die L0-Interoperabilität wird mit Hilfe der in Listing J.4 gezeigten Types-Class (TC) sichergestellt. Da der verhandelbare Satz der GP-Erweiterungen und TLM-Phasen nicht leer ist, wird in der TC die `L1_config` aus Abschnitt 4.6.5 verwendet. Verwendet ein TLM-2.0-Socket die `catlm_AHB_traits` TC, so bedeutet dies, dass der Socket die unverhandelbaren

	GP-Erweiterungen	TLM-Phasen

unverhandelbar	`transfer_type`, `transfer_size`, `burst_type`, `prot_info`, `splitNretry`	`[BEGIN|ABORT|END]_REQ`
		`[BEGIN|ABORT]_BURSTREQ`
		`[BEGIN|END]_DATA`
		`[BEGIN|END]_RESP`
		`[BEGIN|END]_NOTOK`

verhandelbar	`master_id`, `is_locked`, `split_mask`	`[BEGIN|END]_LOCK`
		`[BEGIN|END]_BUSREQ`
		`[BEGIN|END]_GRANT`
		`[BEGIN|END]_SPLIT`

Tabelle I.11.: Verhandelbare und unverhandelbare Sätze von GP-Erweiterungen und TLM-Phasen beim AHB

Sätze voll unterstützt und über die verhandelbaren Sätze Angaben bezüglich der Erfordernisgrade macht.

Die konkreten Angaben bezüglich der Erfordernisgrade hängen vom Modul (Master, Slave oder AHB-Modell), dem Socket (Initiator oder Socket) und den Fähigkeiten des Moduls (Split-Capable Slave oder nicht, AHB-Modell unterstützt Split oder nicht) ab. Listing I.12 zeigt fünf verschiedene Konfigurationen, die den Konfigurationen entsprechen, die in realen Modellen anzutreffen sind. Andere Konfigurationen sind denkbar, wenn z.B. Traffic-Generatoren eingesetzt werden, die nicht alle Transfers unterstützen können oder wollen.

```
1  /*
2  Annahme: Die Konfiguration eines Sockets ist in einem L1_config-Objekt
3  namens m_config gespeichert
4  */
5
6  //Konfiguration fuer einen Bus-Master-Initiatorsocket:
7  //  Arbitrierung ist zwingend
8  //  Split-Information erreicht normale Bus-Master nicht
9  void config_for_bus_master()
10 {
11   m_config.set_phase_exigency(BEGIN_BUSREQ,catlm::catlm_mandatory);
12   m_config.set_phase_exigency(END_BUSREQ,catlm::catlm_mandatory);
13   m_config.set_phase_exigency(BEGIN_GRANT,catlm::catlm_mandatory);
14   m_config.set_phase_exigency(END_GRANT,catlm::catlm_mandatory);
15   m_config.set_phase_exigency(BEGIN_LOCK,catlm::catlm_mandatory);
16   m_config.set_phase_exigency(END_LOCK,catlm::catlm_mandatory);
17   //nicht gelistete Phasen/Erweiterungen sind automatisch abgelehnt
18 }
19
20 //Konfiguration fuer einen AHB-Modell-Initiatorsocket (verbunden mit Bus-Slave):
21 //  Arbitrierung ist nicht vorhanden
22 //  Split-Information ist optional, da sowohl Split-Capable als auch nicht
23 //  Split-Capable Slaves angeschlossen werden koennen
24 void config_for_ahb()
25 {
26   m_config.set_phase_exigency(BEGIN_SPLIT,catlm::catlm_optional);
27   m_config.set_phase_exigency(END_SPLIT,catlm::catlm_optional);
```

```
28
29    m_config.set_extension_exigency(is_locked::ID,catlm::catlm_optional);
30    m_config.set_extension_exigency(master_id::ID,catlm::catlm_optional);
31    m_config.set_extension_exigency(split_mask::ID,catlm::catlm_optional);
32    //nicht gelistete Phasen/Erweiterungen sind automatisch abgelehnt
33  }
34
35  //Konfiguration fuer einen Split-Capable Bus-Slave-Targetsocket:
36  //  Arbitrierung ist nicht vorhanden
37  //  Split-Informationen sind zwingend
38  void config_for_split_capable_bus_slave()
39  {
40    m_config.set_phase_exigency(BEGIN_SPLIT,catlm::catlm_mandatory);
41    m_config.set_phase_exigency(END_SPLIT,catlm::catlm_mandatory);
42
43    m_config.set_extension_exigency(is_locked::ID,catlm::catlm_mandatory);
44    m_config.set_extension_exigency(master_id::ID,catlm::catlm_mandatory);
45    m_config.set_extension_exigency(split_mask::ID,catlm::catlm_mandatory);
46    //nicht gelistete Phasen/Erweiterungen sind automatisch abgelehnt
47  }
48
49  //Konfiguration fuer einen Split-Capable Bus-Slave-Targetsocket:
50  //  Arbitrierung ist nicht vorhanden
51  //  Split-Informationen wird explizit nicht verwendet
52  void config_for_not_split_capable_bus_slave()
53  {
54    m_config.set_phase_exigency(BEGIN_SPLIT,catlm::catlm_rejected);
55    m_config.set_phase_exigency(END_SPLIT,catlm::catlm_rejected);
56
57    m_config.set_extension_exigency(is_locked::ID,catlm::catlm_rejected);
58    m_config.set_extension_exigency(master_id::ID,catlm::catlm_rejected);
59    m_config.set_extension_exigency(split_mask::ID,catlm::catlm_rejected);
60    //nicht gelistete Phasen/Erweiterungen sind automatisch abgelehnt
61  }
62
63  //Konfiguration fuer einen AHB-Modell-Targetsocket (verbunden mit Bus-Master:
64  //  Arbitrierung ist zwingend
65  //  Split-Informationen sind nicht vorhanden
66  void config_for_ahb()
67  {
68    m_config.set_phase_exigency(BEGIN_BUSREQ,catlm::catlm_mandatory);
69    m_config.set_phase_exigency(END_BUSREQ,catlm::catlm_mandatory);
70    m_config.set_phase_exigency(BEGIN_GRANT,catlm::catlm_mandatory);
71    m_config.set_phase_exigency(END_GRANT,catlm::catlm_mandatory);
72
73    //Lock ist optional, da ggf. nicht jeder Master bus-locking verwendet
74    m_config.set_phase_exigency(BEGIN_LOCK,catlm::catlm_optional);
75    m_config.set_phase_exigency(END_LOCK,catlm::catlm_optional);
76    //nicht gelistete Phasen/Erweiterungen sind automatisch abgelehnt
77  }
```

Listing I.12: L1-Konfigurationen für verschiedene AHB-Module

Mit den in Listing I.12 dargestellten Konfigurationen wird sichergestellt, dass ein AHB-Bus-Master nicht direkt mit einem AHB-Bus-Slave verbunden werden kann. Es wird dazwischen immer ein AHB-Modell benötigt. Darüber hinaus erlauben die gezeigten Konfigurationen, dass an ein AHB-Modell sowohl Split-Capable als auch nicht Split-Capable Slaves angeschlossen werden können. Zusätzlich ist es Mastern am AHB erlaubt, auf die Verwendung der LOCK-Phase zu verzichten.

I.13. Zusammenfassung

In diesem Abschnitt (I) wurde detailliert das GP- und TLM-Phase-Mapping des AMBA AHB präsentiert. Die daraus resultierende Implementierung, also der Satz der TLM-Phasen, der GP-Erweiterungen, sowie entsprechender Sockets mit Bindungschecks ist in Anhang J dargestellt. Hier wird noch einmal ein Überblick über die für Nutzer entscheidende Informationen gelistet.

Davon ausgehend, dass die in dieser Arbeit bestimmten Modellierungskonzepte für taktgenaue busphasenbasierte *J-R*-Simulation bekannt, also zum Beispiel als neue Teile in den TLM-2.0-Standard eingeflossen und auch das AHB-Protokoll selbst bekannt sind, benötigt ein Anwender die Implementierung aus Anhang J, die Informationen aus den Tabellen I.2 (Defaultzustände der Masterports), I.3 (Ports der Phasen des Masters), die Beschreibung der Phasenstarts, -enden und -abbrüche (gegebenfalls in Prosa), weiterhin die Tabellen I.4 (Defaultzuständ der Slaveports), I.5 (Ports der Phasen des Slaves), I.6 (GP-Mapping für den Master), I.7 (GP-Mapping des Slaves), I.8 (TLM-Phase-Mapping, aber in einer Post-TLM-Phasenreduktionsvariante), I.9 (Phasenassoziationen, aber in einer Post-TLM-Phasenreduktionsvariante), Abschnitt I.10 und schließlich Tabelle I.11.

Zusätzlich dazu sind noch folgende Hinweise nützlich:

- Ein Schreib-/Lese-Burst wird mit einem GP modelliert. Die erste Phase ist stets `BEGIN_REQ`, die letzte `END_RESP` oder `END_DATA`.
- `[BEGIN|END]_NOTOK` verwenden das GP in dessen Transfer der Fehler auftrat.
- `[BEGIN|ABORT|END]_[BUSREQ|LOCK|GRANT|SPLIT]` haben jeweils ein eigenes GP. Eine Transaktion für das jeweilige Signal besteht immer nur aus den entsprechenden zwei Phasen (z.b. `BEGIN_LOCK` und `ABORT_LOCK`).
- GRANT und SPLIT starten im Target und gehen zum Initiator, benötigen dafür aber eine eigene Transaktion. In diesem Fall ist es zulässig, dass ein Target ein GP instanziiert.
- IDLE-Transfers werden nicht explizit modelliert. Sie entstehen durch die Abwesenheit von `BEGIN_REQ` und `BEGIN_BURSTREQ` in einem Takt, in dem sie auftreten dürften.

Diese Information kann im vorliegenden Format auf circa 10 DIN-A4 Seiten festgehalten werden und besteht darüber hinaus nahezu ausschließlich aus übersichtlichen Tabellen. Die Dokumentation einer vollständig proprietären Implementierung wäre (auch unter der Voraussetzung, dass das AHB-Protokoll bekannt ist) bei Weitem umfangreicher und komplexer.

I.14. Beispiel

Die Nutzung des für den AHB definierten TLM-2.0-Interfaces für die taktgenaue busphasenbasierte *J-R*-Simulation wird mit Hilfe eines einfachen Beispiels illustriert. Ein TLM-Modell

eines Masters möchte mit einem TLM-Modell eines Slaves über ein TLM-AHB-Modell eine Kommunikation durchführen, die der in Abbildung I.13 gezeigten RTL-Kommunikation gemäß der über die Busphasen definierten Inputabstraktion J und der Relevanzauswahl R entspricht. Dann müssen der Master, der AHB und der Slave derart modelliert werden, dass folgende Kommunikationen im TLM-Modell zu beobachten sind[4].

Eingehende Bemerkung : Das Initiator-Socket-Target-Socket-Paar zwischen Master- und AHB-Modell wird im Folgenden nur M-A-Verbindung genannt, das Initiator-Socket-Target-Socket-Paar zwischen AHB- und Slave-Modell wird A-S-Verbindung genannt.
Generic Payloads werden im folgenden über Namen identifiziert, d.h. taucht in einem Aufruf ein bereits verwendeter Variablenname auf, so bezieht sich dieser auf das GP, das bei erster Nennung der Variable deklariert wurde.
Die Werte der TLM-Phasen werden im Folgenden direkt im Funktionsaufruf dargestellt, sind in der Realität aber Werte eines TLM-Phasenobjektes.
Gemäß der in Abschnitt 4.4 getroffenen Empfehlung wird von Zeitannotationen abgesehen (sie ist also stets Null) und sie werden im Beispiel nicht genannt.
Wenn nicht explizit erwähnt, ist der Rückgabewert immer `TLM_ACCEPTED`.

Takt 1 : Auf der M-A-Verbindung wird `nb_transport_fw(gp1,BEGIN_BUSREQ)` aufgerufen. Dies entspricht dem Setzen von HBUSREQ. `gp1` ist dabei ein vom Master erzeugtes nicht weiter initialisiertes GP. Die eigentliche Information besteht in der Übermittlung der TLM-Phase.

Takt 3 : Auf der M-A-Verbindung wird `nb_transport_bw(gp2,BEGIN_GRANT)` aufgerufen. Dies entspricht dem Setzen von HREADY, wenn HGRANT auf Eins ist. `gp2` ist dabei ein vom AHB erzeugtes nicht weiter initialisiertes GP. Die eigentliche Information besteht in der Übermittlung der TLM-Phase.

Takt 4 : Auf der M-A-Verbindung wird `nb_transport_fw(gp3,BEGIN_REQ)` aufgerufen. Dies entspricht dem Start eines NONSEQ-Transfers. `gp3` ist dabei ein vom Master erzeugtes GP, in dem `m_command` auf `TLM_WRITE_COMMAND`, `m_address` auf 16 gesetzt sind und die GP-Erweiterungen `transfer_size`, `burst_type` und `prot_info` die Werte 2, 3 bzw. 1 haben und als gültig markiert sind.
Auf der M-A-Verbindung wird `nb_transport_fw(gp1,ABORT_BUSREQ)` aufgerufen. Dies entspricht dem Rücksetzen von HBUSREQ.

Takt 5 : Auf der A-S-Verbindung wird `nb_transport_fw(gp3,BEGIN_REQ)` aufgerufen. Dies geschieht nicht in Takt 4, da erst jetzt HREADYIN aus Eins ist. Der Inhalte von `gp3` ist unmodifiziert. Dies entspricht dem Start eines NONSEQ-Transfers. Der Rückgabewert muss `TLM_UPDATED` mit der Änderung der TLM-Phase auf `END_REQ` sein, da es sich um eine Einzeltaktphase handelt (siehe Abschnitt 4.9). Dies wird als `nb_transport_bw(gp3,END_REQ)` auf die M-A-Verbindung weitergeleitet. Der Slave kennt jetzt alle Adressen (siehe Listing I.10) und auch die Werte aller weiteren Signale des Addressbusses aller folgenden

[4]Wie dies erreicht werden kann wird für den PLB in Abschnitt 6.3 gezeigt.

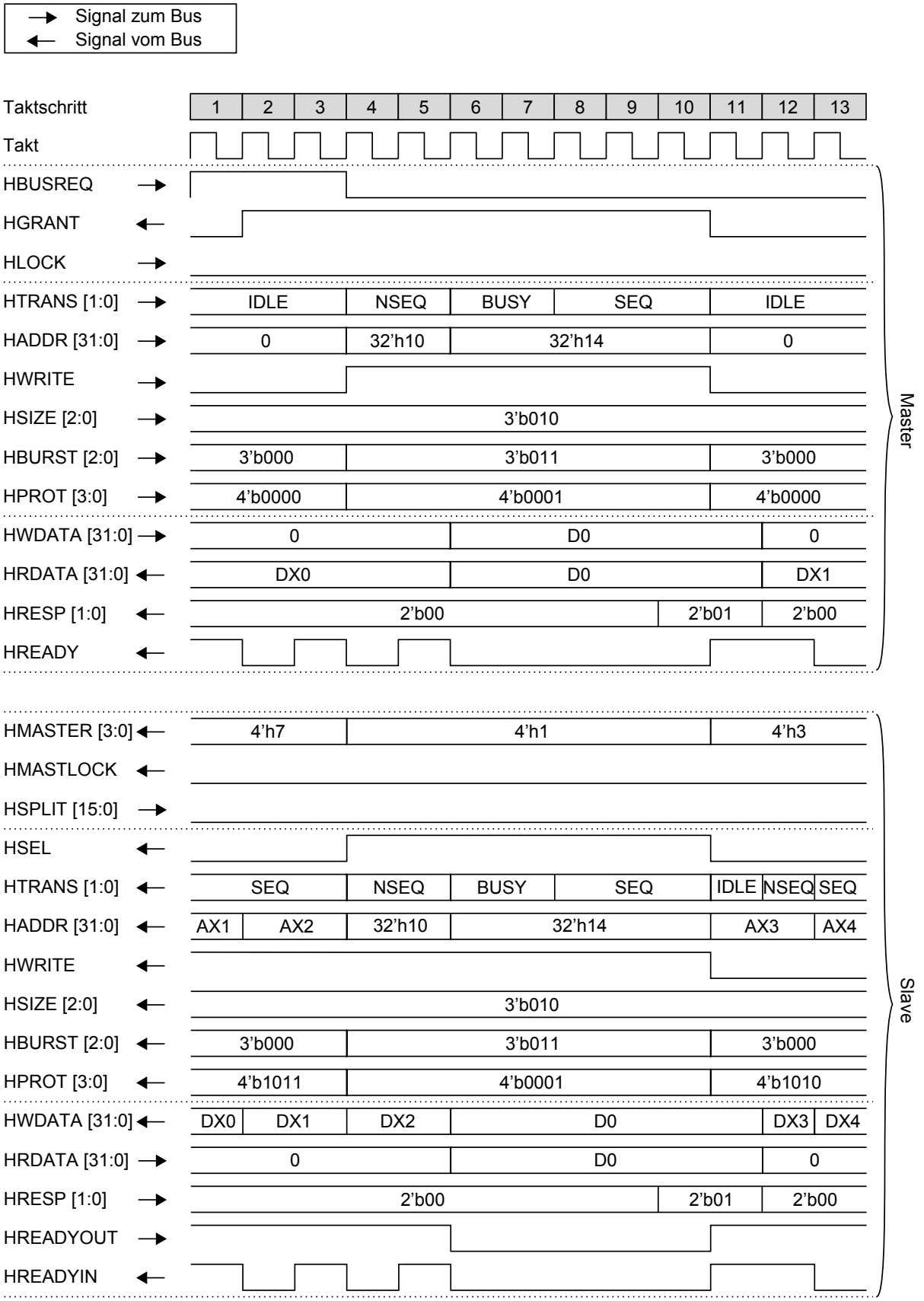


Abbildung I.13.: AHB-Write-Burst mit BUSY-Zyklus, Error und Burstabbruch

SEQ-Transfers. Vom Inhalt von gp3 sind jetzt für ihn nur noch das Datenfeld und die transfer_type-GP-Erweiterung interessant.

Takt 6 : Auf der M-A-Verbindung wird nb_transport_fw(gp3,BEGIN_BURSTREQ) aufgerufen. In gp3 wird die GP-Erweiterung transfer_type als gültig markiert und der Wert wird auf BUSY gesetzt. Dies entspricht dem Start eines BUSY-Transfers. Der Aufruf wird genau so auf die A-S-Verbindung weitergeleitet. Der Slave weiß nun, dass der Master noch nicht mit dem Burst fortfahren kann.

Auf der M-A-Verbindung wird nb_transport_fw(gp3,BEGIN_DATA) aufgerufen. In gp3 werden (wenn nicht bereits geschehen) mindestens die Bytes 0 bis 3 des Datenfeldes entsprechend D0 aus Abbildung I.13 gesetzt. Dieser Aufruf wird so auf die A-S-Verbindung weitergeleitet. Der Slave kennt jetzt also die Daten des ersten Transfers.

Takt 8 : Auf der M-A-Verbindung wird nb_transport_fw(gp3,BEGIN_BURSTREQ) aufgerufen. In gp3 wird die GP-Erweiterung transfer_type als gültig markiert und der Wert wird auf SEQ gesetzt. Dies entspricht dem Start eines SEQ-Transfers. Die BURSTREQ-Phase wird also erneut gestartet (was den alten Start „überschreibt“). Der Aufruf wird genau so auf die A-S-Verbindung weitergeleitet. Der Slave weiß nun, dass sobald er den noch offenen Datenzyklus beendet, im darauffolgenden Takt ein neuer Datenzyklus starten wird.

Takt 10 : Der Slave hat einen Fehler beim Schreibzugriff erkannt. Auf der A-S-Verbindung wird nb_transport_bw(gp3,BEGIN_NOTOK) aufgerufen, dabei wird in gp3 m_response_status auf TLM_GENERIC_ERROR_RESPONSE gesetzt. Der Aufruf wird so auf die M-A-Verbindung weitergeleitet. Der Master weiß nun, dass im nächsten Takt der Datenzyklus enden wird (dies gibt das AHB-Protokoll so vor) und er kann entscheiden, ob er den Burst fortsetzt oder abbricht.

Takt 11 : Der Master entscheidet den Burst abzubrechen. Auf der M-A-Verbindung wird nb_transport_fw(gp3,ABORT_BURSTREQ) aufgerufen. Der Aufruf wird so auf die A-S-Verbindung weitergeleitet. Der Slave weiß damit, dass der Burst vorbei ist.

Auf der M-A-Verbindung wird nb_transport_bw(gp2,ABORT_GRANT) aufgerufen. Dies entspricht dem Rücksetzen von HGRANT.

Auf der A-S-Verbindung werden nb_transport_bw(gp3,ABORT_NOTOK) und nb_transport_bw(gp3,END_DATA) aufgerufen, da einerseits die Zwei-Takt-Error-Antwort beendet ist und andererseits auch der Datenzyklus endet.

Man erkennt, dass die Kommunikation von den Aufrufen von nb_transport dominiert wird und Änderungen von Werten nur selten stattfinden. Darüber hinaus werden die Werte nie kopiert, es wird nur ein Zeiger auf das GP-ausgetauscht. Des Weiteren sind einige Takte (2, 7 und 9) ungenutzt, in denen zum Beispiel das AHB-Modell komplett inaktiv sein kann. Dies erlaubt die effizientere Simulation dieser Kommunikation als in RTL. Darüber hinaus kommunizieren nur ein Master, der AHB und ein Slave; die GPs werden nur zwischen genau diesen Modulen ausgetauscht. Nicht addressierte Slaves empfangen keinerlei Transaktionen, da Phasen der Slaves nur bei HSEL=1 starten können. In einem realen AHB wird die Kom-

munikation immer an alle Slaves gesendet, die erst über die aktive Auswertung von HSEL erkennen, dass sie nicht addressiert sind. Auch dies trägt zu einer effizienteren Simulation bei.

J. Ein TLM-2.0-Interface für taktgenaue J-R-Simulation des AHB

Inhalt

J.1. Einleitung

Dieser Anhang enthält die vollständige Implementierung des TLM-2.0-AHB-Interfaces, wie in es in Abschnitt I konstruiert wurde. Die gesamt Implementierung befindet sich in einer einzigen Datei namens `catlm_AHB.h`.

J.2. #include-Direktiven

Wie in Listing J.1 gezeigt müssen die Dateien `multi_passthrough_initiator_socket.h` und `multi_passthrough_target_socket.h` aus den TLM-2.0-Utilities inkludiert werden. Dabei handelt es sich wohlgemerkt um das entsprechend dieser Arbeit überarbeitete TLM-2.0. Die dritte `#inlcude`-Direktive erlaubt es den Object-Pool aus der Boost-Bibliothek ([Clea06]$_i$) zu nutzen.

```
1 #include "multi_passthrough_initiator_socket.h"
2 #include "multi_passthrough_target_socket.h"
3 #include "boost/pool/object_pool.hpp"
```

Listing J.1: Include-Direktiven für die AHB-Implementierung

J.3. GP-Erweiterungen

Die GP-Erweiterungen verwenden einen Boost-Object-Pool, falls ein GP kopiert werden muss. Ansonsten wird das Memory-Management mit dem in Abschnitt beschriebenen 4.8.4 Menchanismus automatisch ersetzt. Die Implementierung der GP-Erweiterungen findet in den Zeilen 35ff. von Listing J.2 statt.

```
1  template <typename T, typename DERIVED_EXT>
2  struct catlm_extension_base : public tlm::tlm_extension<DERIVED_EXT>
3  {
4  protected:
5     void copy_from(tlm::tlm_extension_base const & ext){
6      const DERIVED_EXT * t=static_cast<const DERIVED_EXT*>(&ext);
7      value=t->value;
8     }
9
10    tlm::tlm_extension_base* clone() const {
11      DERIVED_EXT * t=get_pool()->construct();
12      t->value=value;
13      return t;
14    }
15
16    void free(){
17      get_pool()->free(static_cast<DERIVED_EXT*>(this));
18    }
19
20    static boost::object_pool< DERIVED_EXT >* get_pool(){
21     static boost::object_pool< DERIVED_EXT > s_ptr(5);
22     return &s_ptr;
23    }
24
25 public:
26    static DERIVED_EXT * create_inst(){
27     return get_pool()->construct();
28    }
29
30    catlm_extension_base(){}
31
32    T value;
33 };
34
35 enum transfer_type_enum {BUSY=1, SEQ=3};
36 struct transfer_type: public catlm_extension_base< transfer_type_enum, transfer_type>{};
37 struct transfer_size: public catlm_extension_base<short, transfer_size>{};
38 struct burst_type: public catlm_extension_base<short, burst_type>{};
39 struct prot_info: public catlm_extension_base<short, prot_info >{};
40 struct splitNretry: public catlm_extension_base<bool, splitNretry >{};
41 struct split_mask: public catlm_extension_base<int, split_mask >{};
42 struct master_id: public catlm_extension_base<int, master_id >{};
43 struct is_locked: public catlm_extension_base<bool, is_locked >{};
```

Listing J.2: Implementierung der GP-Erweiterungen für den AHB

J.4. Erweiterte TLM-Phasen

Für den AHB werden zusätzliche TLM-Phasen benötigt und entsprechend der im TLM-2.0-LRM enthaltenen Anweisungen erzeugt (Listing J.3).

```
1  DECLARE_EXTENDED_PHASE(ABORT_REQ);
2  DECLARE_EXTENDED_PHASE(BEGIN_BURSTREQ);
3  DECLARE_EXTENDED_PHASE(ABORT_BURSTREQ);
4  DECLARE_EXTENDED_PHASE(BEGIN_DATA);
5  DECLARE_EXTENDED_PHASE(END_DATA);
6  DECLARE_EXTENDED_PHASE(BEGIN_NOTOK);
7  DECLARE_EXTENDED_PHASE(END_NOTOK);
8  DECLARE_EXTENDED_PHASE(BEGIN_LOCK);
9  DECLARE_EXTENDED_PHASE(END_LOCK);
10 DECLARE_EXTENDED_PHASE(BEGIN_BUSREQ);
11 DECLARE_EXTENDED_PHASE(END_BUSREQ);
12 DECLARE_EXTENDED_PHASE(BEGIN_GRANT);
13 DECLARE_EXTENDED_PHASE(END_GRANT);
14 DECLARE_EXTENDED_PHASE(BEGIN_SPLIT);
15 DECLARE_EXTENDED_PHASE(END_SPLIT);
```

Listing J.3: Implementierung der zusätzlichen TLM-Phasen für den AHB

J.5. Traits-Class

Da für den AHB ein verhandelbarer Satz von GP-Erweiterungen existiert und da der in Abschnitt 4.11 beschriebene Mechanismus zur korrekten Behandlung von kombinatorischen Berechnungen den für die L2-Bindungschecks übermittelten Zeiger auf ein L2-Objekt nutzt, wird eine L1-Konfiguration wie in Abschnitt 4.6.5 beschrieben, benötigt. Somit ergibt sich die TC wie in Listing J.4 gezeigt.

```
1 struct catlm_AHB_traits
2 {
3   typedef tlm::tlm_generic_payload tlm_payload_type ;
4   typedef tlm::tlm_phase tlm_phase_type ;
5   typedef catlm::L1_config tlm_rt_bind_config_type ;
6 } ;
```

Listing J.4: Traits-Class für taktgenaue AHB-Simulation

J.6. Initiator-Socket

Der Initiator-Socket für die AHB-Modellierung (Listing J.5) ist ein erweiterter Multi-Socket aus den TLM-2.0-Utilities. Dies erlaubt es einen Master an mehrere AHB-Modelle anzuschliessen (z.B. ein CPU-Modell an einen Instruction-Side-AHB und einen Data-Side-AHB) und an ein AHB-Modell mehrere Slaves anzuschliessen. Er kann als Socket eines Bus-Masters oder eines AHB-Modells (vgl. Abschnitt I.12) konfiguriert werden. Darüber hinaus unterstützt der Socket die Timing-Information-Distribution (TID), wie in Abschnitt 4.11 erklärt.

```
template<typename MODULE, unsigned int BUSWIDTH>
class AHB_init_socket
  : public tlm_utils::multi_passthrough_initiator_socket<MODULE, BUSWIDTH, catlm_AHB_traits>
{
  typedef tlm_utils::multi_passthrough_initiator_socket<MODULE, BUSWIDTH, catlm_AHB_traits> base_type;
  typedef catlm::initiator_timing_support_base<catlm_AHB_traits> timing_suport_type;

public:
  AHB_init_socket(const char* name)
    : base_type(name)
  {
    //registrieren des get_config-Callbacks
    base_type::register_get_config_bw(this, &AHB_init_socket::get_config) ;
    //registrieren des get_config-Callbacks
    base_type::register_get_name_bw(this, &AHB_init_socket::get_name) ;
    //zuweisen der L1-Konfiguration an den Socket
    base_type::set_rt_bind_config_ptr(&m_config);
    //setzen des L2-Callbacks, der nach erfolgreichem L1-Test durchgefuehrt wird
    m_config.register_L2_callback(&m_timing_support, &timing_suport_type::register_binding);
    //setzen des L2-Objekt, das an die andere Seite zu uebermitteln ist
    m_config.set_L2_ptr(&m_timing_support);
  }

  void config_for_bus_master()
  {
    m_config.set_phase_exigency(BEGIN_BUSREQ,catlm::catlm_mandatory);
    m_config.set_phase_exigency(END_BUSREQ,catlm::catlm_mandatory);
    m_config.set_phase_exigency(BEGIN_GRANT,catlm::catlm_mandatory);
    m_config.set_phase_exigency(END_GRANT,catlm::catlm_mandatory);
    m_config.set_phase_exigency(BEGIN_LOCK,catlm::catlm_mandatory);
    m_config.set_phase_exigency(END_LOCK,catlm::catlm_mandatory);
    //nicht gelistete Phasen/Erweiterungen sind automatisch abgelehnt
  }

  void config_for_ahb()
  {
    m_config.set_phase_exigency(BEGIN_SPLIT,catlm::catlm_optional);
    m_config.set_phase_exigency(END_SPLIT,catlm::catlm_optional);

    m_config.set_extension_exigency(is_locked::ID,catlm::catlm_optional);
    m_config.set_extension_exigency(master_id::ID,catlm::catlm_optional);
    m_config.set_extension_exigency(split_mask::ID,catlm::catlm_optional);
    //nicht gelistete Phasen/Erweiterungen sind automatisch abgelehnt
  }

  void set_timing_listener_callback(MODULE* owner,
                          void(MODULE::*timing_cb) (catlm::timing_info<catlm_AHB_traits>))
  {
    m_timing_support.set_timing_listener_callback(owner, timing_cb);
  }

  void set_initiator_timing(catlm::timing_info<catlm_AHB_traits>& info)
  {
    m_timing_support.set_initiator_timing(info);
  }

  void set_initiator_timing(catlm::timing_info<catlm_AHB_traits>& info, unsigned int index)
  {
    m_timing_support.set_initiator_timing(info, index);
  }
```

```
61
62 protected:
63   //der Callback zur Rueckgabe der Konfiguration
64   const tlm::tlm_rt_bind_config_base* get_config(int){return &m_config;}
65   std::string get_name(int){return this->name();}
66   timing_suport_type m_timing_support;
67   catlm::L1_config m_config;
68 };
```

Listing J.5: Initiator-Socket für AHB-Modellierung

J.7. Target-Socket

Der Target-Socket für die AHB-Modellierung (Listing J.6) ist ein erweiterter Multi-Socket aus den TLM-2.0-Utilities. Dies erlaubt es mehrere Master an ein AHB-Modelle anzuschliessen oder einen Slave an mehrere AHB-Modelle anzuschliessen (z.B. Multi-Port-Memory-Controller). Er kann als Socket eines Bus-Slaves (Split-Capable oder nicht) oder eines AHB-Modells (vgl. Abschnitt I.12) konfiguriert werden. Darüber hinaus unterstützt der Socket die Timing-Information-Distribution (TID), wie in Abschnitt 4.11 erklärt.

```
1  template<typename MODULE, unsigned int BUSWIDTH>
2  class AHB_target_socket
3    : public tlm_utils::multi_passthrough_target_socket<MODULE, BUSWIDTH, catlm_AHB_traits>
4  {
5    typedef tlm_utils::multi_passthrough_target_socket<MODULE, BUSWIDTH, catlm_AHB_traits> base_type;
6    typedef catlm::target_timing_support_base<catlm_AHB_traits> timing_suport_type;
7
8  public:
9    AHB_target_socket(const char* name)
10     : base_type(name)
11   {
12     //registrieren des get_config-Callbacks
13     base_type::register_get_config_fw(this, &AHB_target_socket::get_config) ;
14     //registrieren des get_config-Callbacks
15     base_type::register_get_name_fw(this, &AHB_target_socket::get_name) ;
16     //zuweisen der L1-Konfiguration an den Socket
17     base_type::set_rt_bind_config_ptr(&m_config);
18     //setzen des L2-Callbacks
19     m_config.register_L2_callback(&m_timing_support, &timing_suport_type::register_binding);
20     //setzen des L2-Objekt
21     m_config.set_L2_ptr(&m_timing_support);
22   }
23
24   void config_for_split_capable_bus_slave()
25   {
26     m_config.set_phase_exigency(BEGIN_SPLIT,catlm::catlm_mandatory);
27     m_config.set_phase_exigency(END_SPLIT,catlm::catlm_mandatory);
28
29     m_config.set_extension_exigency(is_locked::ID,catlm::catlm_mandatory);
30     m_config.set_extension_exigency(master_id::ID,catlm::catlm_mandatory);
31     m_config.set_extension_exigency(split_mask::ID,catlm::catlm_mandatory);
32     //nicht gelistete Phasen/Erweiterungen sind automatisch abgelehnt
33   }
34
```

```
35   void config_for_not_split_capable_bus_slave()
36   {
37     m_config.set_phase_exigency(BEGIN_SPLIT,catlm::catlm_rejected);
38     m_config.set_phase_exigency(END_SPLIT,catlm::catlm_rejected);
39
40     m_config.set_extension_exigency(is_locked::ID,catlm::catlm_rejected);
41     m_config.set_extension_exigency(master_id::ID,catlm::catlm_rejected);
42     m_config.set_extension_exigency(split_mask::ID,catlm::catlm_rejected);
43     //nicht gelistete Phasen/Erweiterungen sind automatisch abgelehnt
44   }
45
46   void config_for_ahb()
47   {
48     m_config.set_phase_exigency(BEGIN_BUSREQ,catlm::catlm_mandatory);
49     m_config.set_phase_exigency(END_BUSREQ,catlm::catlm_mandatory);
50     m_config.set_phase_exigency(BEGIN_GRANT,catlm::catlm_mandatory);
51     m_config.set_phase_exigency(END_GRANT,catlm::catlm_mandatory);
52     m_config.set_phase_exigency(BEGIN_LOCK,catlm::catlm_optional);
53     m_config.set_phase_exigency(END_LOCK,catlm::catlm_optional);
54     //nicht gelistete Phasen/Erweiterungen sind automatisch abgelehnt
55   }
56
57   void set_timing_listener_callback(MODULE* owner,
58                       void(MODULE::*timing_cb) (catlm::timing_info<catlm_AHB_traits>))
59   {
60     m_timing_support.set_timing_listener_callback(owner, timing_cb);
61   }
62
63   void set_target_timing(catlm::timing_info<catlm_AHB_traits>& info)
64   {
65     m_timing_support.set_target_timing(info);
66   }
67
68   void set_target_timing(catlm::timing_info<catlm_AHB_traits>& info, unsigned int index)
69   {
70     m_timing_support.set_target_timing(info, index);
71   }
72
73 protected:
74   //der Callback zur Rueckgabe der Konfiguration
75   const tlm::tlm_rt_bind_config_base* get_config(int){return &m_config;}
76   std::string get_name(int){return this->name();}
77   timing_suport_type m_timing_support;
78   catlm::L1_config m_config;
79 };
```

Listing J.6: Target-Socket für AHB-Modellierung

K. Beispiel zur Distribution von Synchronisationsebenen

Inhalt

K.1. Einleitung

Dieser Anhang zeigt mit einem ausführlichen Code-Auszug wie Synchronisationsebenen für den PLB berechnet und die entsprechenden Informationen weitergeleitet werden können. Damit wird die Verwendung der Synchronisationsebenen und der damit möglichen partiellen Prozess-Ausführungsordnung verdeutlicht[1].

K.2. Verwendete Member-Variablen des PLB-Modells

Im Rahmen der Berechnung und Distribution der Synchronisationsebenen (SEs) kommen die folgenden Member-Variablen des Typs `timing_info<PLB_traits>` zum Einsatz:

`m_info_from_masters` : Enthält die höchsten bisher von Mastern empfangenen SEs.
`m_info_from_slaves` : Enthält die höchsten bisher von Slaves empfangenen SEs.
`m_info_to_masters` : Enthält die SEs, die Mastern mitgeteilt werden müssen.
`m_info_to_slaves` : Enthält die SEs, die Slaves mitgeteilt werden müssen.

K.3. Die Timing-Listener-Callbacks

Da der PLB Master und Slaves in verschiedenen Takt-Domänen nicht unterstützt, müssen die Timing-Listener-Callbacks (siehe Abschnitt 4.11 und Abbildung 4.47) die übergebenen `timing_info`-Objekte zuerst auf `is_non_default` überprüfen. Ist dem so, wird eine

[1] In der finalen Implementierung des PLB-Modells wurde die halbautomatische Distribution der Synchronisationsebenen nach [Melz09] verwendet. Dies ändert aber nichts an der hier gezeigten Funktion.

Fehlermedlung ausgegeben. Danach können die übermittelten Informationen bezüglich der Synchronisationsebenen ausgelesen und geeignet verwendet werden.

Die Timing-Listener-Callbacks (für Master die Zeilen 2 bis 17, für Slaves die Zeilen 20 bis 37 in Listing K.1) speichern intern die Timing-Info, die nur die höchsten Synchronisationsebenen, die von Mastern oder Slaves mitgeteilt wurden, enthält. Wenn es eine Änderung gab, werden die Synchronisationsebenen für den PLB mittels `re_calc_sls` neu berechnet.

```
1  //Dieser Callback wird aufgerufen, sobald ein Master eine Timing-Info uebermittelt
2  void PLB::master_timing_listener(timing_info<PLB_traits> info)
3  {
4    if (info.is_non_default()){
5     SC_REPORT_ERROR("PLBv4.6-TLM", "Der PLB kann nicht von Mastern aus anderen Taktdomaenen genutzt werden.");
6    }
7    //nun werden nur Maxima aus der uebergebenen Info entnommen
8    bool change_of_m_info=false;
9    tlm::tlm_phase master_phases[5]={tlm::BEGIN_REQ, ABORT_REQ, BEGIN_RBC, BEGIN_DATA, BEGIN_DBC};
10   for (unsigned int i=0; i<5; i++){
11     if (info.get_call_sync_layer(master_phases[i])> m_info_from_masters.get_call_sync_layer(master_phases[i])){
12       m_info_from_masters.set_call_sync_layer(master_phases[i], info.get_call_sync_layer(master_phases[i]));
13       change_of_m_info=true;
14     }
15   }
16   if (change_of_m_info) re_calc_sls();
17 }
18
19 //Dieser Callback wird aufgerufen, sobald ein Slave eine Timing-Info uebermittelt
20 void PLB::slave_timing_listener(timing_info<PLB_traits> info)
21 {
22   if (info.is_non_default()){
23     SC_REPORT_ERROR("PLBv4.6-TLM", "Der PLB kann nicht von Slaves aus anderen Taktdomaenen genutzt werden.");
24   }
25   //nun werden nur Maxima aus der uebergebenen Info entnommen
26   bool change_of_m_info=false;
27   tlm::tlm_phase slave_phases[12]={tlm::END_REQ, tlm::BEGIN_RESP, PLB::BEGIN_RBT, PLB::END_DATA,PLB::BEGIN_DBT,
28     PLB::BEGIN_WAIT, PLB::BEGIN_RCOMP, PLB::BEGIN_DCOMP, PLB::BEGIN_IRQ, PLB::ABORT_IRQ, PLB::BEGIN_BUSY,
29     PLB::ABORT_BUSY};
30   for (unsigned int i=0; i<12; i++){
31     if (info.get_call_sync_layer(slave_phases[i])> m_info_from_slaves.get_call_sync_layer(slave_phases[i])){
32       m_info_from_slaves.set_call_sync_layer(slave_phases[i], info.get_call_sync_layer(slave_phases[i]));
33       change_of_m_info=true;
34     }
35   }
36   if (change_of_m_info) re_calc_sls();
37 }
```

Listing K.1: Timing-Listener-Callbacks im PLB-Modell

K.4. Die Berechnung der Synchronisationsebenen

Die in Listing K.2 gezeigten Funktionen, insbesondere die Funktion `re_calc_sls` tragen Sorge für die korrekte Bestimmung der Synchronisationsebenen. Die ausführlichen Begründungen für die entsprechenden Abhängigkeiten der Synchronisationsebenen wurden bereits in Abschnitt 6.3.2 gegeben, im Folgenden werden lediglich die Berechnungen kurz erläutert.

Die Synchronisationsebene `breq_sl` ist eine Ebene höher als die spätest mögliche `ABORT_REQ`-, `BEGIN_DATA`- oder `BEGIN_DBC`-Phase (Zeile 6 in Listing K.2; alle weiteren Zeilenreferenzen in diesem Unterabschnitt beziehen sich auf das gleiche Listing).

Die Synchronisationsebene `abrt_sl` für den `ereq_handler`-Prozess ist eine Ebene höher als die spätest mögliche `ABORT_REQ`- oder `END_REQ`-Phase (Zeile 10).

Die Synchronisationsebenen `rprim_sl` bzw. `dprim_sl` liegen eine Ebene über dem spätesten `END_REQ` und entsprechendem COMP (Zeilen 13 und 17). Für `dprim_sl` wird auch noch die höchste Synchronisationsebene von `BEGIN_DATA` herangezogen.

Die Synchronisationsebenen `bsy_sl` bzw. `irq_sl` liegen eine Ebene über dem spätesten `[BEGIN|ABORT]_[BUSY|IRQ]` (Zeilen 20 und 21).

Die Synchronisationsebenen `pnd_sl` (Zeile 22) liegt eine Ebene über dem spätesten `BEGIN_REQ`.

Nachdem die Ebenen berechnet sind, wird überprüft, ob es zu Änderungen in den Informationen für die Master oder Slaves gekommen ist (Zeilen 24 bis 94). Ist dem so, werden die Master und Slaves benachrichtigt (Zeile 96f.).

```
1  unsigned int PLB::get_max(unsigned int a, ...); //liefert das Maximum der Argumente
2
3  //diese Funktion berechnet alle notwendigen Synchronisationsebenen und leitet Infos weiter
4  void PLB:: re_calc_sls(){
5    //Berechnung der Synchronisationsebenen
6    unsigned int new_breq_sl=get_max(
7      m_info_from_masters.get_call_sync_layer(ABORT_REQ),
8      m_info_from_masters.get_call_sync_layer(BEGIN_DATA),
9      m_info_from_masters.get_call_sync_layer(BEGIN_DBC))+1;
10   unsigned int new_abrt_sl=get_max(
11     m_info_from_masters.get_call_sync_layer(ABORT_REQ),
12     m_info_from_slaves.get_call_sync_layer(tlm::END_REQ))+1;
13   unsigned int new_dprim_sl=get_max(
14     m_info_from_slaves.get_call_sync_layer(BEGIN_DCOMP),
15     m_info_from_slaves.get_call_sync_layer(tlm:: END_REQ),
16     m_info_from_masters. get_call_sync_layer(BEGIN_DATA))+1;
17   unsigned int new_rprim_sl=get_max(
18     m_info_from_slaves.get_call_sync_layer(BEGIN_RCOMP),
19     m_info_from_slaves.get_call_sync_layer(tlm:: END_REQ))+1;
20   unsigned int new_bsy_sl= m_info_from_slaves.get_call_sync_layer(BEGIN_BSY)+1;
21   unsigned int new_irq_sl= m_info_from_slaves.get_call_sync_layer(BEGIN_IRQ)+1;
22   unsigned int new_pnd_sl= m_info_from_masters.get_call_sync_layer(BEGIN_REQ)+1;
23   //Aktualisieren der Info fuer Master
24   bool send_to_master=false;
25   if (m_info_to_masters.get_call_sync_layer(tlm::END_REQ)<new_abrt_sl){
26     abrt_sl=new_abrt_sl;
27     m_info_to_masters.set_call_sync_layer(tlm::END_REQ, abrt_sl);
28     send_to_master=true;
29   }
30   if (m_info_to_masters.get_call_sync_layer(BEGIN_BSY)<new_bsy_sl){
31     bsy_sl=new_bsy_sl;
32     m_info_to_masters.set_call_sync_layer(BEGIN_BSY, bsy_sl);
33     send_to_master=true;
34   }
35   if (m_info_to_masters.get_call_sync_layer(BEGIN_IRQ)<new_irq_sl){
36     irq_sl=new_irq_sl;
37     m_info_to_masters.set_call_sync_layer(BEGIN_IRQ, irq_sl);
38     send_to_master=true;
39   }
40
41   tlm::tlm_phase slave_flow_thru_phases[4]={tlm::BEGIN_RESP, PLB::BEGIN_RBT, PLB::END_DATA, PLB::BEGIN_DBT};
42   for (unsigned int i=0; i<4; i++){
43     if (m_info_from_slaves.get_call_sync_layer(slave_flow_thru_phases[i])>
```

```
44        m_info_to_masters.get_call_sync_layer(slave_flow_thru_phases[i]){
45        m_info_to_masters.set_call_sync_layer(slave_flow_thru_phases[i],
46          m_info_from_slaves.get_call_sync_layer(slave_flow_thru_phases[i]));
47        send_to_master =true;
48      }
49    }
50
51    //Aktualisieren der Info fuer Slaves
52    bool send_to_slave=false;
53    //BEGIN_REQ, BEGIN_DATA und BEGIN_DBC haben alle den gleichen SL
54    if (m_info_to_slaves.get_call_sync_layer(tlm::BEGIN_REQ)<new_breq_sl){
55      breq_sl=new_breq_sl;
56      m_info_to_slaves.set_call_sync_layer(tlm::BEGIN_REQ, breq_sl);
57      m_info_to_slaves.set_call_sync_layer(BEGIN_DATA, breq_sl);
58      m_info_to_slaves.set_call_sync_layer(BEGIN_DBC, breq_sl);
59      send_to_slave=true;
60    }
61    if (m_info_to_slaves.get_call_sync_layer(ABORT_REQ)<new_abrt_sl){
62      abrt_sl=new_abrt_sl;
63      m_info_to_slaves.set_call_sync_layer(ABORT_REQ, abrt_sl);
64      send_to_slave=true;
65    }
66    if (m_info_to_slaves.get_call_sync_layer(BEGIN_DPRIM)<new_dprim_sl){
67      dprim_sl=new_dprim_sl;
68      m_info_to_slaves.set_call_sync_layer(BEGIN_DPRIM, dprim_sl);
69      send_to_slave=true;
70    }
71    if (m_info_to_slaves.get_call_sync_layer(BEGIN_RPRIM)<new_rprim_sl){
72      rprim_sl=new_rprim_sl;
73      m_info_to_slaves.set_call_sync_layer(BEGIN_RPRIM, rprim_sl);
74      send_to_slave=true;
75    }
76    if (m_info_to_slaves.get_call_sync_layer(BEGIN_RBC)<m_info_from_masters.get_call_sync_layer(BEGIN_RBC)){
77      m_info_to_slaves.set_call_sync_layer(BEGIN_RBC, m_info_from_masters.get_call_sync_layer(BEGIN_RBC));
78      send_to_slave=true;
79    }
80
81    //Jetzt die Pending Info
82    if (new_pnd_sl>pnd_sl){
83          m_info_to_slaves.set_call_sync_layer(BEGIN_DPND, new_pnd_sl);
84          m_info_to_slaves.set_call_sync_layer(ABORT_DPND, new_pnd_sl);
85          m_info_to_slaves.set_call_sync_layer(BEGIN_RPND, new_pnd_sl);
86          m_info_to_slaves.set_call_sync_layer(ABORT_RPND, new_pnd_sl);
87          m_info_to_masters.set_call_sync_layer(BEGIN_DPND, new_pnd_sl);
88          m_info_to_masters.set_call_sync_layer(ABORT_DPND, new_pnd_sl);
89          m_info_to_masters.set_call_sync_layer(BEGIN_RPND, new_pnd_sl);
90          m_info_to_masters.set_call_sync_layer(ABORT_RPND, new_pnd_sl);
91          pnd_sl=new_pnd_sl;
92          send_to_slave=true;
93          send_to_master=true;
94    }
95    //jetzt senden wenn noetig
96    if (send_to_master) target_socket.set_target_timing(m_info_to_masters);
97    if (send_to_slave) init_socket.set_initiator_timing(m_info_to_slaves);
98  }
```

Listing K.2: Timing-Listener im PLB-Modell

L. Beispiel zur Überprüfung der Taktgenauigkeit des PLB v4.6-Modells

Inhalt

L.1. Einleitung

Dieser Anhang stellt in Abbildung L.1 auf Seite 285 auf gegenüberliegenden Seiten die Waveform-Diagramme der RTL- und TLM-Simulation[1] des PLB v4.6 bei (gemäß J und R) identischen Stimuli gegenüber.

L.2. Testfall

Die Master im in Abbildung L.1 auf Seite 285 gezeigten Testfall erzeugten die in Tabelle L.1 gezeigten Stimuli. Dabei machte keiner der Master eine Pause zwischen dem Ende einer Transaktion und dem Beginn der nächsten Transaktion.

Modul	Transaktionsart	Transaktionsanzahl	Burstlänge	Priorität
Master 0	Read	4	4	2
Master 1	Read	2	16	1
Master 2	Write	3	6	3
Master 3	Write	8	1	2

Tabelle L.1.: Von den Mastern erzeugte Stimuli für das Beispiel zur Bestimmung der Taktgenauigkeit

[1] Das TLM-Modell wurde mit der Option zum VCD-Tracing ausgestattet.

Die Slaves erzeugten dabei die in Tabelle L.2 auf der nächsten Seite gezeigten Verzögerungen.

Modul	Request-Accept-Verzögerung	Lese- bzw. Schreibdatenverzögerung
Slave 0	0	1
Slave 1	2	0

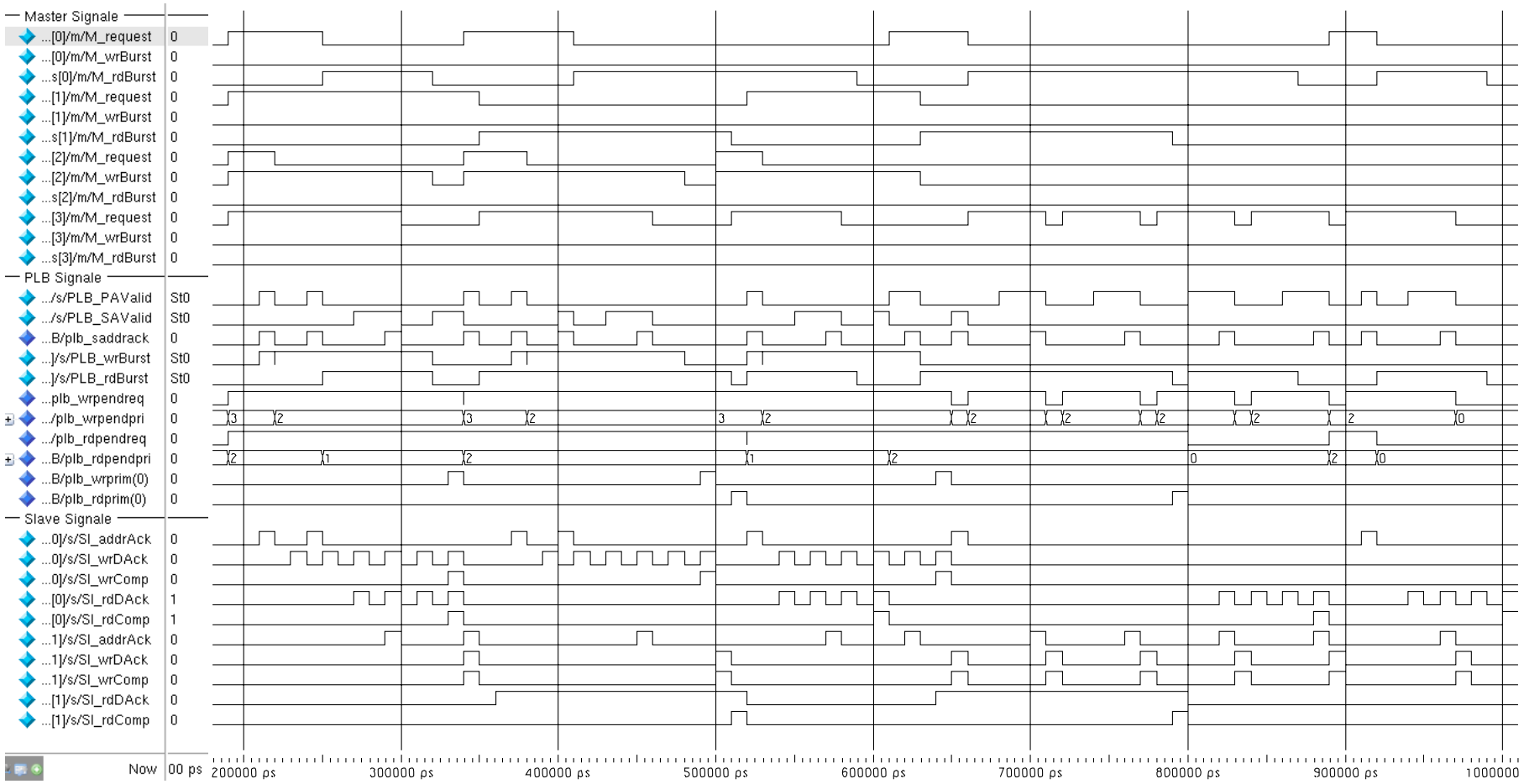
Tabelle L.2.: Von den Slaves erzeugte Verzögerungen im Beispiel zur Bestimmung der Taktgenauigkeit

L.3. Auswertung

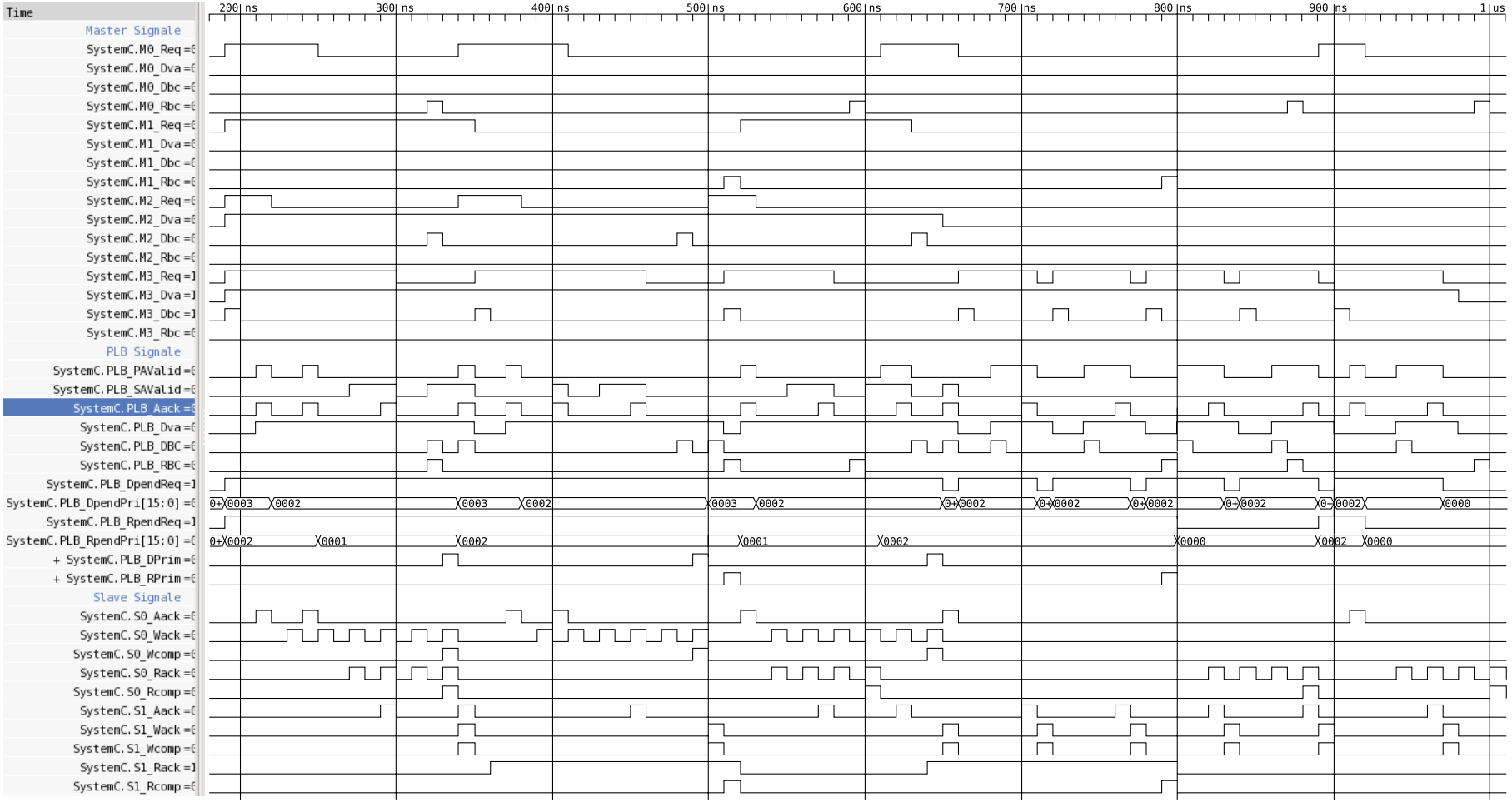
Der in Abbildung L.1 auf Seite 285 gezeigte Fall illustriert die korrekte Behandlung von nach Read und Write getrenntem Address-Pipelining, die Promotion von sekundären Requests und die Verteilung der Information über anhängige Requests. Die Taktperiode, die für die Simulation verwendet wurde, betrug 10 ns. Man erkennt, dass die Busphasen, die in Abschnitt 6.2 definiert wurden, in Abbildung L.1a und b zu den gleichen Zeitpunkten beziehungsweise Takten starten, enden bzw. abbrechen.

Um die Dauer der Datenphase im TLM-Modell besser erkennen zu können, ist im VCD-Trace ein Signal DVA erkennbar, welches immer dann auf eins ist, wenn eine Datenphase läuft. Man kann auch erkennen, dass aufgrund der Art und Weise der Definition der Busphasen von den Signalen `wrburst` und `rdburst` abstrahiert wird. Lediglich ihre für den Slave nicht vorhersagbarer Wechsel auf Null[2] werden durch DBC bzw. RBC markiert.

[2] Alle Transaktionen sind Bursts; selbst die 1-Wort-Transaktionen.



(a) RTL; Screen-Shot von Mentor Graphics Modelsim [MtGr08]



(b) TLM; Screen-Shot von GTKWave [Bybe10]$_i$

Abbildung L.1.: Ergebnis der RTL-Simulation des PLB mit vier Mastern und zwei Slaves. Master 0: vier 4-Wort-Read-Bursts an Slave 0 mit Priorität 2. Master 1: zwei 16-Wort-Read-Bursts an Slave 1 mit Priorität 1. Master 2: drei 6-Wort-Write-Bursts an Slave 0 mit Priorität 3. Master 3: acht 1-Wort-Write-Bursts an Slave 1 mit Priorität 2. Kein Master macht Pausen zwischen Transfers. Slave 0: Keine Request- aber einen Takt Datenverzögerung. Slave 1: Zwei Takte Request- aber keine Datenverzögerung.

Verzeichnisse

Abkürzungsverzeichnis

AMBA Advanced-Microcontroller-Bus-Architecture

AT Approximately Timed

AXI Advanced-eXtensible-Interface

BA Bus-Accurate

BP Base-Protocol

BTI Blocking-Transport-Interface

CCATB Cycle-Count-Accurate-at-Transaction-Boundaries

CT Cycle-Timed

DES diskreter Event-Simulator

DMA Direct-Memory-Access

DMI Direct-Memory-Interface

DTI Debug-Transport-Interface

GP Generic Payload

IMC Interface-Method-Call

IP Intellectual Property

ISS Instruction-Set-Simulator

LRM Language-Reference-Manual

LT Loosely Timed

MM Memory-Manager

MMB memory-mapped Bus

MMBIF memory-mapped Bus-Interface

MMIO Memory-Mapped-Input-Output

NBTI Non-blocking-Transport-Interface

NoC Network-on-Chip

OCP Open-Core-Protocol

OPB On-Chip-Peripheral-Bus

OSCI Open-SystemC-Initiative

PPAO partielle Prozess-Ausführungsordnung

PLB Processor-Local-Bus

PMB Processor-Memory-Bus

PV Programmer's View

RTL Register-Transfer-Level

RQ Runnable-Queue

SLD System-Level-Design

SoC System-on-Chip

SPMB Shared-Processor-Memory-Bus

TC Types-Class

TID Timing-Information-Distribution

TLM Transaction-Level-Modellierung

WG Working-Group

Stichwortverzeichnis

U

Z

Literaturverzeichnis

[Aldi06] Aldis, James: Use of SystemC Modelling in Creation and Use Of a SOC Platform: Experiences and Lessons learnt from OMAP-2. In: *Platform Based Design at the Electronic System Level*. Springer, Dordrecht, Netherlands : Springer, 2006, S. 31–47

[ARM-99] ARM Ltd.: *AMBA Specification (Rev. 2.0)*. Cambridge, UK : ARM Ltd., 1999

[ARM-03] ARM Ltd.: *AMBA AXI Protocol v1.0*. Cambridge, UK : ARM Ltd., 2003

[Aziz09] Aziz, Syed M.: A cycle-accurate transaction level SystemC model for a serial communication bus. In: *Computers & Electrical Engineering* 35 (2009), Nr. 5, 790 - 802. http://dx.doi.org/DOI:10.1016/j.compeleceng.2008.11.029. – DOI DOI: 10.1016/j.compeleceng.2008.11.029. – ISSN 0045–7906

[BAGK07] Burton, Mark ; Aldis, James ; Günzel, Robert ; Klingauf, Wolfgang: Transaction Level Modeling: A Reflection on what TLM is and how TLMs may be defined. In: *Forum on Design Languages 2007, Barcelona* (2007), September

[BaMP07] Bailey, Brian ; Martin, Grant ; Piziali, Andrew: *ESL Design and Verification*. San Francisco, USA : Morgan Kaufmann, 2007

[BCNN04] Banks, Jerry ; Carson, John ; Nelson, Barry L. ; Nicol, David: *Discrete-Event System Simulation, Fourth Edition*. Upper Saddle River, NJ, USA : Prentice Hall, 2004

[Bode07] Bode, Christian: *Eine generische Interface-Beschreibung einfacher IP-Cores in XML*, Technische Universität Braunschweig, Studienarbeit, 2007

[BuGr07] Buchmann, Richard ; Greiner, Alain: A fully static scheduling approach for fast cycle accurate systemC simulation of MPSoCs. In: *Microelectronics, 2007. ICM 2007. Internatonal Conference on*, 2007, S. 101 –104

[BuMo06] Burton, Mark (Hrsg.) ; Morawiec, Adam (Hrsg.): *Platform Based Design at the Electronic System Level*. Springer, Dordrecht, Netherlands : Springer, 2006. – 31–47 S.

[CaGa03] Cai, Lucai ; Gajski, Daniel: Transaction level modeling: an overview. In: *Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on.* Newport Beach, USA, Oct. 2003, S. 19–24

[CCC+03] Caldari, Marco ; Conti, Massimo ; Coppola, Marcello ; Crippa, Paolo ; Orcioni, Simone ; Pieralisi, Lorenzo ; Turchetti, Claudio: System-Level Power Analysis Methodology Applied to the AMBA AHB Bus. In: *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe.* Washington, DC, USA : IEEE Computer Society, 2003, S. 20032

[CCG+03] Coppola, Marcello ; Curaba, Stephane ; Grammatikakis, Miltos ; Maruccia, Giuseppe ; Papariello, Francesco: *On-Chip Communication Network: User Manual V1.0.1.* France/Greece : ST Microelectronics/ISD S.A., 2003

[CCH+99] Chang, Henry ; Cooke, Larry ; Hunt, Merrill ; Martin, Grant ; McNelly, Andrew ; Todd, Lee: *Surviving the SOC Revolution: A Guide to Platform-Based Design.* Norwell, MA, USA : Kluwer Academic Publishers, 1999

[DBD+06] Dhanwada, Nagu ; Bergamaschi, Reinaldo ; Dungan, William ; Nair, Indira ; Gramann, Paul ; Dougherty, William ; Lin, Ing-Chao: Transaction-level modeling for architectural and power analysis of PowerPC and CoreConnect-based systems. In: *Design Automation for Embedded Systems* 10 (2006), September, Nr. 2-3, 105–125. `http://dx.doi.org/10.1007/s10617-006-9586-7`. – DOI 10.1007/s10617–006–9586–7. – ISSN 0929–5585

[Donl04] Donlin, Adam: Transaction level modeling: flows and use models. In: *Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004. International Conference on.* Stockholm, Sweden, Sept. 2004, S. 75–80

[Edwa07] Edwards, Chris: Modelling standard edges closer to settling interface woes. In: *Electronics* 5 (2007), Aug.-Sep., Nr. 4, S. 6–7. – ISSN 1754–1778

[Ghen05] Ghenassia, Frank (Hrsg.): *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems* . Dordrecht, The Netherlands. : Springer, 2005

[GHJV94] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John M.: *Design Patterns: Elements of Reusable Object-Oriented Software.* illustrated edition. Reading, MA, USA : Addison-Wesley Professional, 1994

[Golz10] Golze, Ulrich: *Skript zur Vorlesung Chip- und System-Entwurf II.* Braunschweig : Technische Universität Braunschweig, Abteilung E.I.S., 2010

[Grel08] Grellier, Tierrie: Extending SystemC clocks to model SoC. In: *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on.* Stuttgart, Deutschland, Sept. 2008, S. 13 –18

[GrGS05] Grabbe, Cornelia ; Grüttner, Kim ; Schubert, Thorsten: Specification of Hardware/Software Communication Design Methodology based on Abstract Communication Models. In: *ICODES Project Report, European Commission* (2005), Nr. ICODES/OFFIS/R/D11/1.1

[Grot02] Grotker, Thorsten: *System Design with SystemC.* Norwell, MA, USA : Kluwer Academic Publishers, 2002

[GüKA07] Günzel, Robert ; Klingauf, Wolfgang ; Aldis, James: Combinatorial Dependencies in Transaction Level Models. In: *Proc. FDL07.* Barcelona, Spain, September 2007

[Günz07] Günzel, Robert: *Towards a Natively NoC Capable OCP Implementation*, OCP SLD WG intern, Technischer Report, Mai 2007

[Günz08] Günzel, Robert: Automatic Integration of Non-Bus Hardware IP into SoC-Platforms for Use by Software. In: *Embedded and Ubiquitous Computing, 2008. EUC '08. IEEE/IFIP International Conference on* Bd. 1. Shanghai, China, Dec. 2008, S. 59 –65

[Günz10a] Günzel, Robert: *Embecosm Application Notes 1 and 2 and MacOS 10.4*, Technische Universität Braunschweig, Technischer Report, Abteilung E.I.S., 2010. `http://chschroeder.gamiro.de/rg/or1ksim_macOS10.4.pdf`, Abruf: 17. November 2010

[Günz10b] Günzel, Robert: *Erweiterungen von TLM-2.0 zur taktgenauen Modellierung: Implementierung und Experimente*, Technische Universität Braunschweig, Technischer Report, Abteilung E.I.S., 2010

[GZD+00] Gajski, Daniel D. ; Zhu, Jianwen ; Doemer, Rainer ; Gerstlauer, Andreas ; Zhao, Shuqing: *SpecC: Specification Language and Methodology.* Norwell, MA, USA : Kluwer Academic Publishers, 2000

[IBM-99] IBM Corp.: *CoreConnect Bus Architecture Data Sheet.* Hopewell Junction, NY, USA : IBM Corp., 1999

[IBM-01] IBM Corp.: *On-Chip Peripheral Bus: Architecture Specifications v2.1.* Research Triangle Park, NC, USA : IBM Corp., 2001

[IBM-04] IBM Corp.: *Processor Local Bus: Architecture Specifications v4.6.* Research Triangle Park, NC, USA : IBM Corp., 2004

[IBM-07] IBM Corp.: *Processor Local Bus: Architecture Specifications v4.7*. Hopewell Junction, NY, USA : IBM Corp., 2007

[IEEE05] IEEE Computer Society: IEEE Standard for SystemVerilog- Unified Hardware Design, Specification, and Verification Language. In: *IEEE Std 1800-2005* (2005), S. 1–648

[IEEE06a] IEEE Computer Society: IEEE Std 1364 -2005 IEEE Standard for Verilog Hardware Description Language. In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), S. 1–560

[IEEE06b] IEEE Computer Society: IEEE Std 1666 - 2005 IEEE Standard SystemC Language Reference Manual. In: *IEEE Std 1666-2005* (2006), S. 1–423

[IEEE09] IEEE Computer Society: IEEE Standard VHDL Language Reference Manual. In: *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)* (2009), S. c1–626. `http://dx.doi.org/10.1109/IEEESTD.2009.4772740`. – DOI 10.1109/IEEESTD.2009.4772740

[ISO-03] International Organization for Standardization: Programming Language C++. In: *ISO/IEC 14882:2003* (2003), S. 1–757

[JeWo05] Jerraya, Ahmed A. ; Wolf, Wayne: *Multi-Processor Systems-On-chip*. San Francisco, USA : Morgan Kaufmann, 2005

[Josu99] Josuttis, Nicolai M.: *The C++ Standard Library: A Tutorial and Reference*. Reading, MA, USA : Addison-Wesley Professional, 1999

[KeRi88] Kernighan, Brian W. ; Ritchie, Dennis: *The C Programming Language*. Second. Upper Saddle River, NJ, USA : Prentice Hall, 1988

[KGB+06] Klingauf, Wolfgang ; Günzel, Robert ; Bringmann, Oliver ; Parfuntseu, Pavel ; Burton, Mark: GreenBus - a generic interconnect fabric for transaction level modelling. In: *Design Automation Conference, 2006 43rd ACM/IEEE*, 2006. – ISSN 0738–100X, S. 905–910

[Klin08] Klingauf, Wolfgang: *Systematic Transaction Level Communication Modeling with SystemC*, Technical University Braunschweig, Diss., February 2008

[KMN+00] Keutzer, Kurt ; Malik, Sharad ; Newton, Richard ; Rabaey, Jan ; Sangiovanni-Vincentelli, Alberto: System Level Design: Orthogonalization of Concerns and Platform-Based Design. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19 (2000), S. 1523–1543

[Krue09] Krüsselin, Matthias: *Entwicklung eines Simulationsframeworks für dynamisch rekonfigurierbare Systeme auf Basis der SPARC Architektur*, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, Diplomarbeit, 2009

[Lapl04] Laplante, Phillip A.: *Real-Time Systems Design and Analysis*. Third. Piscataway, NJ, USA : Wiley-IEEE Press, 2004

[LeYi07] Lee, Junghee ; Yi, Joonhwan: Industrial experience with cycle error computation of cycle-accurate transaction level models. In: *SOC Conference, 2007 IEEE International*. Hsinchu, Taiwan, Sept. 2007, S. 155–158

[Libe99] Liberty, Jesse: *The complete idiot's guide to a career in computer programming*. Indianapolis, USA : Alpha Books, 1999

[Lin-06] Lin, Youn-Long: Essential Issues In System-On-A-Chip Design. In: *Essential Issues In SoC Design*. Dordrecht, Netherlands : Springer, 2006, S. 1–5

[LoTs09] Lo, Chen K. ; Tsay, Ren S.: Automatic generation of Cycle Accurate and Cycle Count Accurate transaction level bus models from a formal model. In: *ASP-DAC '09: Proceedings of the 2009 Asia and South Pacific Design Automation Conference*. Piscataway, NJ, USA : IEEE Press, 2009, S. 558–563

[MCRB09] Moll, HWM van ; Corporaal, Henk ; Reyes, Victor ; Boonen, Marleen: Fast and Cycle Accurate Protocol Specific Bus Modeling Using TLM 2.0. In: *DATE '09: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA : IEEE Computer Society, 2009

[Melz09] Melzer, Lennart: *Eine SystemC-2.2-Kernelerweiterung zur Unterstützung kombinatorischer und taktsynchroner TLM-2.0-Kommunikation*, Technische Universität Braunschweig, Bachelorarbeit, 2009

[MtGr08] Mentor Graphics: *ModelSim SE User's Manual: Software Version 6.4a*. Wilsonville, USA : Mentor Graphics, 2008

[MWG+07] Mair, Hugh ; Wang, Alice ; Gammie, Gordon ; Scott, David ; Royannez, Philippe ; Gururajarao, Sumanth ; Chau, Minh ; Lagerquist, Rolf u. a.: A 65-nm Mobile Multimedia Applications Processor with an Adaptive Power Management Scheme to Compensate for Variations. In: *VLSI Circuits, 2007 IEEE Symposium on*. Kyoto,Japan, 2007, S. 224 –225

[PaDB04] Pasricha, Sudeep ; Dutt, Nikil ; Ben-Romdhane, Mohamed: Extending the transaction level modeling approach for fast communication architecture exploration. In: *Design Automation Conference, 2004. Proceedings. 41st*. San Diego, USA, 2004. – ISSN 0738–100X, S. 113–118

[PaHe08] Patterson, David A. ; Hennessy, John L.: *Computer Organization and Design: The Hardware/Software Interface*. Fourth. San Francisco, USA : Morgan Kaufmann, 2008

[Park08] Parker, Philip M.: *Accuracy, Webster's Quotations, Facts and Phrases*. San Diego, CA, USA : ICON Group International, Inc., 2008

[Pedr01] Pedram, Massoud: Power optimization and management in embedded systems. In: *ASP-DAC '01: Proceedings of the 2001 Asia and South Pacific Design Automation Conference*. New York, NY, USA : ACM, 2001, S. 239–244

[RoSa97] Rowson, James A. ; Sangiovanni-Vincentelli, Alberto: Interface-based design. In: *DAC '97: Proceedings of the 34th annual Design Automation Conference*. New York, NY, USA : ACM, 1997, S. 178–183

[ScLM06] Scheffer, Louis ; Lavagno, Luciano ; Martin, Grant: *EDA for IC System Design, Verification, and Testing (Electronic Design Automation for Integrated Circuits Handbook)*. Boca Raton, FL, USA : CRC Press, Inc., 2006

[SCP+03] Sayinta, Ali ; Canverdi, Gorkem ; Pauwels, Marc ; Alshawa, Amer ; Dehaene, Wim: A Mixed Abstraction Level Co-Simulation Case Study Using SystemC for System on Chip Verification. In: *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA : IEEE Computer Society, 2003, S. 20095

[SiMu01] Siegmund, Robert ; Mueller, Dietmar: SystemC-SV: An Extension of SystemC for Mixed Multi-Level Communication Modeling and Interface-Based System Design. In: *Proc. Design, Automation and Test in Europe Conference (DATE)* (Munich, 2001), S. 26–33

[STM-07] ST Microelectronics: *STBus Communication System: Concepts And Definitions - User Manual (Rev. 1)*. Geneva, Switzerland : ST Microelectronics, 2007

[SuDF06] Sutherland, Stuart ; Davidmann, Simon ; Flake, Peter: *Systemverilog For Design: A Guide To Using Systemverilog For Hardware Design And Modeling*. New York, USA : Springer, 2006

[VDK+08] Varma, Ankush ; Debes, Eric ; Kozintsev, Igor ; Klein, Paul ; Jacob, Bruce: Accurate and fast system-level power modeling: An XScale-based case study. In: *ACM Trans. Embed. Comput. Syst.* 7 (2008), Nr. 3, S. 1–20. `http://dx.doi.org/http://doi.acm.org/10.1145/1347375.1347378`. – DOI http://doi.acm.org/10.1145/1347375.1347378. – ISSN 1539–9087

[WiHO06] Wild, Thomas ; Herkersdorf, Andreas ; Ohlendorf, Rainer: Performance evaluation for system-on-chip architectures using trace-based transaction level simulation. In: *DATE '06: Proceedings of the conference on Design, automation and test in Europe*. 3001 Leuven, Belgium, Belgium : European Design and Automation Association, 2006, S. 248–253

[XILI10] XILINX: *LogiCore IP Datasheet Processor Local Bus v4.6 v1.05a*. San Jose, USA : Xilinx Inc., 2010

[ZRRJ04] Zhong, Lin ; Ravi, Srivaths ; Raghunathan, Anand ; Jha, Niraj K.: Power estimation for cycle-accurate functional descriptions of hardware. In: *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*. Washington, DC, USA : IEEE Computer Society, 2004, S. 668–675

Internetquellenverzeichnis

[Ansl09] Ansley, John: Using TLM-2.0 Extensions for Bus Locking and Snooping - Video Tutorial. (2009). http://media.systemc.org/tlm20extensions/index.html, Abruf: 23. Juni 2010

[ARM-07] ARM Ltd.: Cycle Accurate Simulation Interface (CASI). (2007). http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0359b/Cjhfhgdg.html, Abruf: 28. Februar 2011

[ARM-10] ARM Ltd.: Cortex-A8 Processor Documentation. (2010). http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.cortexa.a8/index.html, Abruf: 6. Dezember 2010

[Arte09] Arteris: NoC Solution Product Backgrounder. (2009). www.arteris.com, Abruf: 07. Juli 2009

[Benn09] Bennett, Jeremy ; OpenCores.org (Hrsg.): *Or1ksim v0.3.0 User Guide*. Version: 2009. http://opencores.org/openrisc,downloads, Abruf: 14. Februar 2010

[Benn10] Bennett, Jeremy ; Embecosm Limited (Hrsg.): *Building a Loosely Timed SoC Model with OSCI TLM 2.0*. Version: 2010. http://www.embecosm.com/download/ean1.html, Abruf: 17. November 2010

[Bres06] Brescia Jr., Ralph J.: *United States Patent 7039748: Memory mapped I/O bus selection*. Version: May 2006. http://www.freepatentsonline.com/7039748.html, Abruf: 07 Juli 2009

[Bybe10] Bybell, Tony: GTKWave 3.3 Wave Analyzer User's Guide. (2010). http://gtkwave.sourceforge.net/gtkwave.pdf, Abruf: 24. November 2010

[CaGr10] Carbon Design Systems ; GreenSocs Ltd.: AMBA Modeling Kit. (2010). https://portal.carbondesignsystems.com/ALLIp.aspx?Category=Free%20Downloads, Abruf: 1. März 2010

[Clea06] Cleary, Stephen: Boost Pool Library Version 1.43. (2006). http://www.boost.org/doc/libs/1_43_0/libs/pool/doc/index.html, Abruf: 8. Juli 2010

[CoWa09] CoWare inc.: SystemC Modeling Library Source Code Kit. (2009). http://www.coware.com/solutions/scml_kit.php, Abruf: 22. Februar 2010

[DoGG02] Doemer, Rainer ; Gerstlauer, Andreas ; Gajski, Daniel: SpecC Language Reference Manual. (2002). http://www.cecs.uci.edu/~specc/reference/SpecC_LRM_20.pdf, Abruf: 23. Juni 2009

[Engb08] Engblom, Jakob: SCDSource article: Why virtual platforms need cycle-accurate models. (2008). http://www.scdsource.com/article.php?id=266, Abruf: 25. Juni 2009

[Green07] GreenSocs Initiative: GreenBus Project Web Page. (2007). http://www.greensocs.com/de/projects/GreenBus, Abruf: 26. April 2010

[Herv02] Herveille, Richard ; OpenCores.org (Hrsg.): *System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*. Version: 2002. http://www.opencores.org/projects.cgi/web/wishbone/wbspec_3b.pdf, Abruf: 12. Februar 2008

[IBM-06] IBM Corp.: IBM PowerPC 405 Evaluation Kit with CoreConnect SystemC TLMs. (2006). http://www.ibm.com/developerworks/power/pek, Abruf: 20. Juli 2007

[KBGG07] Klingauf, Wolfgang ; Burton, Mark ; Günzel, Robert ; Golze, Ulrich: SOCcentral featured article: Why We Need Standards for Transaction-Level Modeling. (2007). http://www.soccentral.com/results.asp?EntryID=22293, Abruf: 25. Juni 09

[Klug10] Klug, Brian ; AnandTech, Inc. (Hrsg.): *Two OMAP 3430 Phones: Nokia N900 and Motorola Droid*. Version: 2010. http://www.anandtech.com/show/3764/two-omap-3430-phones-nokia-n900-and-motorola-droid, Abruf: 6. Dezember 2010

[KoHA05] Kogel, Tim ; Haverinen, Anssi ; Aldis, James ; OCP-IP (Hrsg.): *OCP TLM for Architectural Modeling*. Version: 2005. http://www.ocpip.org/socket/whitepapers, Abruf: 23. Juni 2009

[Lamp06] Lampret, Damjan u. a. ; OpenCores.org (Hrsg.): *OpenRISC 1000 Architectural Manual*. Version: 2006. http://opencores.org/svnget,or1k?file=/trunk/docs/openrisc_arch.pdf, Abruf: 17. November 2010

[Maxf08] Maxfield, Clive ; EE Times (Hrsg.): *Altera's new 40nm FPGAs - 2.5 billion transistors!* Version: 2008. http://www.eetimes.com/electronics-products/fpga-pld-products/4104287/

Altera-s-new-40nm-FPGAs--2-5-billion-transistors-, Abruf: 6. Dezember 2010

[MEDB95] Maher III, Robert D. ; Eitrheim, John ; Dunlap, Fred ; Brightman, Thomas B.: *United States Patent 5420989: Coprocessor interface supporting I/O or memory mapped communications.* Version: May 1995. http://www.freepatentsonline.com/5420989.html, Abruf: 07 Juli 2009

[Moto10] Motorola Inc.: Droid by Motorola: Tech Specs. (2010). http://www.motorola.com/Consumers/US-EN/Consumer-Product-and-Services/Mobile-Phones/ci.Motorola-DROID-US-EN.alt, Abruf: 6. Dezember 2010

[Noki10] Nokia: Nokia N900 Technische Daten. (2010). http://www.nokia.de/produkte/mobiltelefone/nokia-n900/technische-daten, Abruf: 6. Dezember 2010

[OCP-06] OCP-IP: Open Core Protocol Specification V2.2. (2006). www.ocpip.org, Abruf: 25. Juni 2009

[OCP-08] OCP-IP: Open Core Protocol TLM Channels r2.2.1. (2008). www.ocpip.org, Abruf: 25. Juni 2009

[OCP-09] OCP-IP: OCP Modeling Kit 2.2x2.1. (2009). www.ocpip.org, Abruf: 6. Oktober 2010

[OSCI00] Open SystemC Initiative: SystemC User's Guide Version 1.1. (2000). http://www.systemc.org, Abruf: 25. Juni 09

[OSCI02] Open SystemC Initiative: Functional Specification for SystemC 2.0. (2002). http://www.systemc.org, Abruf: 25. Juni 09

[OSCI09a] Open SystemC Initiative: OSCI TLM-2.0 Language Reference Manual. (2009). http://www.systemc.org, Abruf: 11. November 09

[OSCI09b] Open SystemC Initiative: Supporting Industry Quotes: Open SystemC Initiative Announces Completion of New Standard Enabling the Real-World Interoperability of Transaction Level Models. (2009). http://www.systemc.org/news/pr/view?item_key=f46f43d71ae3446e9a5514f439437478010dfe49, Abruf: 25. Juni 2009

[PoVR10] PowerVR: SGX Series5 Graphics IP Core Family. (2010). http://www.imgtec.com/powervr/sgx_series5.asp, Abruf: 6. Dezember 2010

[RSPF05] Rose, Adam ; Swan, Stuart ; Pierce, John ; Fernandez, Jean-Michel ; Open SystemC Initiative (Hrsg.): *TLM-1.0 White Paper: Transaction*

Level Modeling in SystemC. Version: 2005. http://www.systemc.org, Abruf: 25. Juni 09

[Rusu09] Rusu, Stefan ; Intel Corporation (Hrsg.): *Nehalem-EX: a 45nm, 8-core Enterprise Processor*. Version: 2009. www.ewh.ieee.org/r6/scv/ssc/Oct2009.pdf, Abruf: 6. Dezember 2010

[ScBr06] Schnieringer, Martin ; Brand, Kevin ; Vast Systems Technology (Hrsg.): *SystemC: Key Modeling Concepts Besides TLM to Boost your Simulation Performance*. Version: 2006. www.vastsystems.com, Abruf: 22. Februar 2010

[Soni08] Sonics Inc.: SonicsLX SMART Interconnect Solution. (2008). http://www.sonicsinc.com/uploads/pdfs/SLX_datasheet_041808.pdf, Abruf: 07. Juli 2009

[STMi05] ST Microelectronics: TAC: Transaction Accurate Communication. (2005). http://www.greensocs.com/TACPackage, Abruf: 6. Dezember 2010

[Swan01] Swan, Stuart ; Open SystemC Initiative (Hrsg.): *An Introduction to System Level Modeling in SystemC 2.0*. Version: 2001. http://www.systemc.org, Abruf: 25. Juni 09

[Texa10] Texas Instruments: OMAP-3430 Overview. (2010). http://focus.ti.com/general/docs/wtbu/wtbuproductcontent.tsp?templateId=6123&navigationId=12643&contentId=14649, Abruf: 6. Dezember 2010

[Weid06] Weidendorfer, Josef: Callgrind, Part of the Valgrind Tool Suite. (2006). http://valgrind.org, Abruf: 8. Juli 2010

Lebenslauf

Name:	Robert Günzel
Geboren:	Am 16. März 1979 in Lauchhammer
Familienstand:	Verheiratet

Schulbildung

1985 bis 1988	Polytechnische Oberschule (POS) *Arthur Wölk* in Senftenberg
1988 bis 1990	POS *Felix Dserschinski* in Stendal
1990 bis 1997	Gymnasium *Johann-Joachim-Winckelmann* in Stendal
04.07.1997	**Abitur**
1997 bis 1999	Marineoperationsschule Bremerhaven
27.09.1999	**Facharbeiter für Kommunikationselektronik (Funktechnik)**

Studium

2000 bis 2005	Studium der Informations-Systemtechnik an der TU Braunschweig
13.05.2005	**Dipl.-Ing. Informations-Systemtechnik** Abschluss *Mit Auszeichnung* Titel der Diplomarbeit: Rapid Prototyping von transaktionsbasiert modellierten Hardware-Software-Systemen mit SystemC Referenten: Prof. Ulrich Golze und Prof. Rolf Ernst
2011	Promotionsgesuch zum Dr.-Ing.

Berufserfahrung: Akademisch

2001 bis 2005	studentische Hilfskraft (HiWi) Abteilung Entwurf integrierter Schaltungen (Prof. U. Golze) TU Braunschweig
2005 bis 2011	wissenschaftlicher Mitarbeiter (WiMi) Abteilung Entwurf integrierter Schaltungen (Prof. U. Golze) TU Braunschweig

Berufserfahrung: Nicht akademisch

1999 bis 2000	Unteroffizier der Deutschen Marine Marinefliegerhorst Nordholz elektische/elektronische Instandhaltung von Langstrecken-Seeaufklärern des Typs Breguet Atlantic